The World According To GARP

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027

Roy Campbell, Steven Goering, Susan Hinrichs,
Brenda Jackels, Joe Loyall, Simon M. Kaplan
University of Illinois
Department of Computer Science
Urbana, IL 61801

June 1988
(revised December 1988)

CUCS-351-88

Abstract

This technical report consists of two papers describing the GARP concurrent programming system. *Garp: Graph Abstractions for Concurrent Programming* investigates construction of dynamic process topologies in parallel processing languages. It proposes the use of a graph-grammar based formalism to control the complexities arising from trying to program such dynamic networks. *Garp: A Graphical Language for Concurrent Programming* describes the GARP system, a programming environment that implements this graph-grammar approach, and gives solutions to example problems in which the topologies of concurrent systems dynamically change.

Prof. Kaiser is supported in part by grants from AT&T, IBM, Siemens and Sun, in part by the Center of Advanced Technology and by the Center for Telecommunications Research, and in part by a DEC Faculty Award. Prof. Kaplan is supported in part by a grant from AT&T Corporation.

Garp : Graph Abstractions for Concurrent Programming

Simon M. Kaplan*
University of Illinois
Department of Computer Science
Urbana, IL 61081

Gail E. Kaiser†
Columbia University
Department of Computer Science
New York, NY 10027

Abstract: Several research projects are investigating parallel processing languages where dynamic process topologies can be constructed. Failure to impose abstractions on interprocess connection patterns can result in arbitrary interconnection topologies that are difficult to understand. We propose the use of a graph-grammar based formalism to control the complexities arising from trying to program such dynamic networks.

keywords: abstraction, actors, concurrency, distributed system, graph grammar, message passing, object-oriented system, parallel processing

There is a growing need for effective ways to organize ... distributed programs [14].

1 Introduction

Languages with the ability to generate arbitrary networks of processes are increasingly a focus of research. Little effort has been directed, however, towards abstractions of the resulting topologies; failure to support such abstractions can lead to chaotic programs that are difficult to understand and maintain. We propose graph grammar-based abstractions as a means for imposing structure on topologies. This paper introduces GARP (Graph Abstractions for Concurrent Programming), a notation based on graph grammars [11] for describing dynamic interconnection topologies.

Graph grammars are similar to string grammars, except that (1) the body of a production is a graph and (2) the rewriting action is the replacement of a vertex by a graph.

*Netmail: kaplan@a.cs.uiuc.edu. Supported in part by a grant from the AT&T Corporation.

†Netmail: kaiser@cs.columbia.edu. Supported in part by grants from the AT&T Foundation, Siemens Research and Technology Laboratories, and New York State Center for Advanced Technology — Computer & Information Systems, and in part by a Digital Equipment Corporation Faculty Award.

The purpose of GARP is to replace arbitrary dynamic communication patterns with abstractions in the same sense that Dijkstra [6] replaced goto-ridden spaghetti code with structured control constructs. There is a cost to this, of course. Just as there are some sequential programs that are difficult to write in a programming language without gotos, there are topologies that are difficult if not impossible to specify using GARP. There is, however, a major difference between the graph grammar approach taken in GARP and the adding of structured programming constructs to sequential programming languages: In the latter case, a fixed set of constructs are always used, while in the former we know that we *need* abstractions, but not what specific patterns to provide. So in GARP the grammar is used to give a set of patterns for a particular program, not for all programs.

GARP uses graph grammars as follows. For a graph generated from a graph grammar, each vertex is interpreted as a process (which we call an *agent*). Agents have *ports* through which they can send and receive messages. Edges in the graph provide asynchronous communications paths between ports. Rewriting of an agent by a production corresponds to the spawning of a graph of new processes as defined in the body of the production, and connecting these into the process topology to replace the agent being rewritten using a connection strategy specified in the production. The agents perform all computation (including the initiation of rewrites on the graph) while the graph grammar acts as an abstraction structure that describes the legal process topologies.

To illustrate the use of the GARP framework we adopt a model in which GARP agents are Scheme [19] programs augmented with port operations (definable in terms of core scheme and a *bag* data type) and operations to control rewriting (defined in terms of graph grammar theory). We emphasize that this model of agents is not central to our use of graph grammars to control process topology complexities; our ideas are equally applicable to other proposals for process models, including Actors [2], Cantor [4], NIL [20] and Argus [14].

Section 2 defines the agents component of GARP, section 3 defines graph grammars and section 4 shows how graph grammars are adapted into the GARP programming formalism. Section 5 discusses the Scheme implementation of our ideas and illustrates GARP with two examples. Section 6 summarizes GARP in the light of the examples. Section 7 compares GARP to related work, especially Actor systems and other applications of graph grammars to distributed systems.

let m = a message (contents are irrelevant)
b = a bag. The internal representation for the Message Handler.
[] = an empty bag
operations
(M-receive b m) ⇒ b ← (⊙ m b)
(M-empty b) ⇒ (if (= b []) *true* *false*)
(M-send b) ⇒ (choice b) and b ← (rest b)
end

Figure 1: Semantics for Message Handlers

2 Message Handlers, Ports and Agents

Computation in GARP is performed by groups of *agents*. Agents communicate among themselves by writing messages to or reading messages from *ports*. Messages written on ports are stored by a *message handler* until read by another agent.

A message handler represents the pool of messages that have been sent to it, but not yet delivered to any agent, as a *bag*. If a is an item that can be inserted into a bag and b and c are bags, the operations on bags are: (⊙ a b) (insertion), (∈ a b) (membership), (= b c) (equality), (choice b), (which nondeterministically chooses an element of b) and (rest b) (which returns the remainder of the bag after a choose). Manna and Waldinger [15] give a theory of bags.

Message handlers are an abstraction built on top of bags. The operations on message handlers, together with their semantics, are given in figure 1. These operations are atomic. An additional level of detail is needed if sending a message to a port is to be a *broadcast* operation; this is a simple extension and the details are omitted.

Agents communicate by reading from and writing to ports. They can be implemented in any language, but must support the following minimal set of porthandling constructs (with behaviour in terms of the message handling commands in figure 1:

- (send port message) is interpreted as (M-recieve port message).
- (msg? port) is interpreted as (not (M-empty port)).
- (on port body) is interpreted as wait until (msg? port) is true, then apply body to the result of (M-send port).

With these operations, more sophisticated operations can be defined, such as:

- `(on-and portlist body)`. Wait until each port in the `portlist` has a message, and then apply the body to the message(s).
- `(on-or ({(port body)}*))`. Nondeterministically choose a port with a message, and apply the corresponding body to the message.
- Looping versions of on, on-or and on-and.

An agent can be thought of as a closure whose parameters include the ports through which it will communicate with other agents, and is similar to a process in CSP [10] or NIL, an actor in Actor Systems, an object in Cantor, a guardian in Argus or a task in Ada[1] [1]. As in Actor Systems, Cantor and NIL, communication among agents is asynchronous, and the arrival order of messages at a port is nondeterministic. By *asynchronous* we mean that the sending process does not know the state of the intended reciever, as opposed to a *synchronous* communication, in which the reciever must be ready and willing to recieve a message before the sender can transmit it.

The interconnections among agents are determined using the graph grammar formalism described in the following section.

3 Graph Grammars

Graph grammars are similar in structure to string grammars. There is an alphabet of symbols, divided into three (disjoint) sets called the terminals, nonterminals and portsymbols. Productions have a nonterminal symbol as the goal (the same nonterminal may be the goal of many productions), and the right-hand side of the production has two parts: a graph (called the bodygraph) and an embedding rule. Each vertex in the bodygraph is labeled by a terminal or nonterminal symbol, and has associated with it a set of portsymbols. Any portsymbol may be associated with many terminals or nonterminals.

The rewriting action on a graph (the host graph) is the *replacement* of a vertex labeled with a nonterminal by the bodygraph of a production for which that nonterminal is the goal, and the *embedding* of the bodygraph into the host graph. This embedding process involves connecting (ports associated with) vertices in the bodygraph to (ports associated with) vertices in the host graph. The embedding process is restricted so that when a vertex v is rewritten, only vertices that are in the *neighborhood* of v—those connected to v by a path of unit length—can be connected to the vertices in the bodygraph that replaces v.

[1] Ada is a trademark of the United States Government, Ada Joint Program Office.

Because we use these graph grammars as an abstraction construct for concurrent programming, we call them concurrent abstraction grammars (CAGs).

Each symbol in the alphabet of terminals and nonterminals has associated with it a set of symbols called *portsymbols*. The same portsymbol may be associated with several terminals or nonterminals. We denote terminals and nonterminals by uppercase characters $X, Y, \cdots$ and portnames by Greek characters $\alpha, \beta, \cdots$. Vertices are denoted $v, w, \cdots$ and the symbol labeling a vertex v is identified by Lab_v. PS_X denotes the set of portsymbols associated with the (terminal or nonterminal) symbol X.

For any graph G, let V_G denote the vertices in G and E_G the edges of G. Each vertex v can be qualified by the portsymbols in PS_{Lab_v} to form a *port-identifier*. Edges are denoted by pairs of port-identifiers, for example $(v.\alpha, w.\beta)$. For any vertex v in a graph G, the neighborhood of v, $\mathcal{N}_v$, is $\{w \mid (v, w) \in E_G\}$.

Definition 1 *A concurrent abstraction graph grammar is a tuple $CAG = (N, T, S, P, Z)$, where N is a finite set of symbols called the nonterminals of the grammar, T is a finite set of symbols called the terminals of the grammar and S is a finite set of symbols called the portsymbols of the grammar such that $T \cap N = N \cap S = T \cap S = \emptyset$; P is a set of productions, where productions are defined in definition 2 below; and Z is a unique distinguished nonterminal known as the axiom of the grammar.*

The axiom Z is the goal of exactly one production and may not appear in any bodygraph. This requirement is not a restriction in practice as one can always augment a grammar with a distinguished production that satisfies this requirement.

Definition 2 *A production in a CAG is defined as: $p : L_p \rightarrow B_p, F_p$ where p is a unique label; $L_p \in N$ is called the goal of the production; B_p is an arbitrary graph (called the bodygraph of the production), where each vertex is labeled by an element of $T \cup N$; and F_p is the embedding rule of the production: a set of pairs $(X.\alpha, L_p.\gamma)$ or $[X.\alpha, Y.\beta]$, where X labels a vertex in $B_p, \alpha \in PS_X, \beta \in PS_Y, \gamma \in PS_{L_p}$.*

The same symbol may appear several times in a bodygraph; this is resolved by subscripting the symbol with an index value to allow them to be distinguished [22].

Definition 3 *The rewriting (or refinement) of a vertex v in a graph G constructed from a CAG by a production p for which Lab_v is the goal is performed in the following steps:*

- *The neighborhood $\mathcal{N}_v$ is identified.*
- *The vertex v and all edges incident on it are removed from G.*
- *The bodygraph B_p is instantiated to form a daughter-graph, which is inserted into G.*

- *The daughter graph is embedded as follows. For each pair in F_p of the form $(X.\alpha, L_p.\gamma)$ an edge is placed from the α port of each vertex in the daughter-graph labeled by X to whatever $v.\gamma$ was connected to before the start of the rewriting. For each pair in F_p of the form $[X.\alpha, Y.\beta]$ an edge is placed from the α port of each vertex in the daughter-graph labeled by X to the β port of each vertex in the set $\{w \mid w \in \mathcal{N}_v \text{ and } Lab_w = Y\}$.*

Note there are two ways to specify an embedding pair, using () or [] notation. The former is often more convenient, but more restrictive as it gives no way to take a port-identifier with several inputs and split those over the vertices in the bodygraph when rewriting.

The most important property that CAGs should have is *confluence*. Such a property would mean that any vertices in the graph can be rewritten in parallel. Unfortunately, we will prove that two vertices that are in one another's neighborhoods cannot be rewritten in parallel (although the graphs are otherwise confluent). This important result means that the rewriting action must be atomic. We approach the proof of this result in two steps: first we prove an intermediate result about the restriction of the extent of embeddings; the limited confluence result follows.

Definition 4 *By* recursive rewriting *of a vertex v we mean possibly rewriting v to some graph—the instantiation of the bodygraph B_p of some rule p for which v is the goal—and then rewriting recursively the vertices in that graph.*

Definition 5 *For any vertex v in a graph G, let $\mathcal{N}_v^*$ denote the universe of possible neighbourhoods of v that could arise by rewriting (recursively) the vertices of $\mathcal{N}_v$; G_v^* denote the universe of graphs obtainable by all possible recursive rewritings of v; and let $S_v^* = G_v^* - (G - \{v\})$,[2] i.e., S_v^* is just the set of subgraphs constructable from v in the recursive rewriting.*

Lemma 6 *Given a vertex v in a graph G, any (recursive) rewriting of v will not introduce edges from the vertices of the daughter graph of v (or any daughter graph recursively introduced into that daughter graph) to any vertex that is not in $\mathcal{N}_v^* \cup S_v^*$.*

Proof: By induction on the rewriting strategy.

Basis: Consider a graph G with a nonterminal vertex v. Refine v by a production p for which Lab_v is the goal. By definition of CAGs, all the vertices in G to which the vertices of the daughter-graph may be connected are in N_v. Therefore the base case does not contradict the theorem.

Inductive Step: Consider now the graph G' with a vertex v', where G' has been formed from G by a

[2] Note this is set difference so the "-" does not distribute.

series of refinements (starting with a vertex v), and v' has been introduced into the graph by one of these refinements. $N_{v'}$ will include only vertices introduced into G by the refinement(s) from v to v', and vertices in N_v^*. Now rewrite v'. Only vertices in $N_{v'}$ can receive edges as the result of embedding the new daughter-graph, so the statement of the theorem remains true under the effect of the rewriting. This completes the proof.

□

Theorem 7 *Two vertices v and w in each other's neighbourhood (i.e. $v \in \mathcal{N}_w$ and $w \in \mathcal{N}_v$) may not be rewritten in parallel.*

Proof: Suppose that it were possible to rewrite the two vertices in parallel and that any rewrite of w would introduce a new vertex x such that $Lab_w = Lab_x$, that would connect to v by the embedding rule, and *vice versa*. Suppose further that once the daughter-graph replacing w hs been instantiated, but before the edge to v has been placed, the rewriting of v begins by removing v from the graph. Clearly at this point there is no vertex v to which to perform the embedding. Therefore it cannot be possible to rewrite two vertices that are in one another's neighbourhoods in parallel.

□

Corollary 8 *Given a graph G constructed from a CAG, the vertices in G may be rewritten in any order.*

Proof: Follows from previous theorem and lemma.

□

4 Relating Graph Grammars and Agents

A GARP program has two parts: a CAG and code for each agent. Vertices in the graph grammar represent agents. Each agent name is either a terminal or nonterminal symbol of the grammar. We extend the reportoire of the agents to include a `rewrite` operation with form:

```
(rewrite name exp ...)
```

where `name` is the label of a production that has the name of the agent about to be rewritten as goal and the `exp ...` are parameters to the production. The interpretation of this operation is the definition of rewriting given in section 3. The `rewrite` action must be the agent's last, because the model of rewriting requires that the agent be replaced by the agents in the bodygraph of the production used in the rewriting.

We extend the production labels of graph grammars to have a list of formal parameters. Each element of the list is a pair `<agent, parameter>`, which identifies the agent in the bodygraph of the production to

which the parameter must be passed, and the specific formal parameter for that agent that should be used. When rewriting, the agents specified in the parameter list are passed the appropriate actual parameter when they are created. This ability to pass arguments from an agent to the agents that replace it provides a way to pass the state of the agent to its replacements. This feature is not unique to our agent system and can be found in Actors and Cantor.

5 Examples

This section of the paper illustrates the use of GARP with two examples written in GARP/Scheme, a version of GARP that uses Scheme as the underlying language for agents. In this system, agents and productions are implemented as first-class scheme objects; we can therefore experiment with parallel programming and our ideas on process struture while retaining all the advantages of a small but extremely powerful programming language[3]. All the features of a programming language required for GARP agents have been implemented in Scheme using that language's powerful macro facilities to provide rewrite rules into core Scheme. There is nothing about the implementation that is unique to Scheme, however; another implementation using the object-oriented language MELD [13] as an underlying framework is under development.

The first example gives a GARP program for quicksort as a tutorial: this is not the most efficient way to sort a stream of numbers, but the GARP program is easy to understand. The second example demonstrates a systems application: this GARP program takes as input an encoding of a dataflow program, generates a dataflow machine tailored for it, and then executes it. This could be useful in allocating processors in a large MIMD machine to dataflow tasks.

The graph grammar for the quicksort example is found in figure 2, and the code for the agents in figure 3. There are two further agents, not shown, modeling standard input and output. This program takes a stream of numbers from standard input, sorts them using divide and conquer, and then passes the result to standard output. The program executes recursively: When the `sort-abs` agent receives a message that is not the end-of-file object, it rewrites itself to a `sort-body` bodygraph, passing the message just read as a seed value to the `split` vertex introduced in the rewrite. This `split` vertex passes all values received by it that are greater than the seed through the `hi` port, all other values through the `lo` port, and the seed itself through the `seedport`. The `join` agent waits for messages on its `lo`, `hi` and `seed` ports and passes the concatenation of these three messages to its `out` port. The `lo` and `hi` ports are connected to `sort-abs` agents, which in turn rewrite themselves to `sort-body` graphs on receipt of an appropriate message. When end-of-file is

[3] A copy of this implementation is available from the first author.

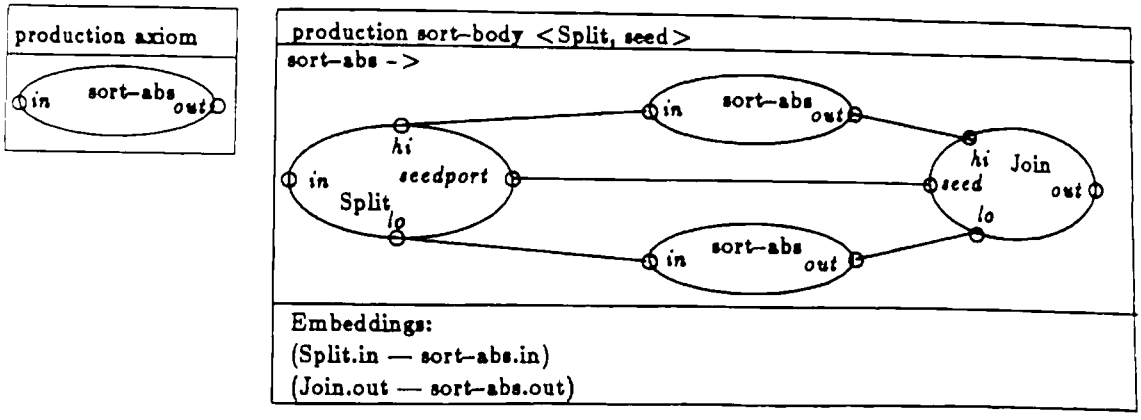


Figure 2: Sort Example – Graph Grammar

encountered, it is propagated through the graph and the `sort-abs` agents left in the graph send the empty list as a "result" value. The `axiom` production starts the program running.

The graph grammar for the dataflow example is given in figure 4 (the agent code is omitted due to space restrictions). In this system, the `controller` agent reads dataflow programs as input messages, and passes them to the `prog` node, which immediately rewrites itself to a new `prog`, an `out-handler` and a `df` agent. The new `prog` waits for another dataflow program, while the `df` node rewrites itself to a dataflow machine (using the `arithmetic`, `identity` and `if-statement` productions, and the `out-handler` waits for the output from the dataflow machine. At each stage of this construction process the `df` agent looks at its `program` parameter, decides what sort of construct to rewrite to, breaks up the `program` accordingly, and passes the components to the new agents via the parameters of the production used in the rewriting. For example, when an arithmetic operation is identified, the `program` can be broken up into the operation, a "left" program fragment and a "right" program fragment. The `df` agent recognises these components and rewrites itself using the `arithmetic` production, passing the operation to the `arithop` agent and the "left" and "right" program fragments to the appropriate new agents. Leaf values, such as constants, are handled internally by the `df` agent (and therefore are in no production explicitly).

Only a simple dataflow language is supported in this example; extension to more complex constructs is not difficult. Once the dataflow machine has been built, it executes the program for which it was constructed and then passes the results to the `out-handler` agent.

GARP is also particularly well suited to the large class of *adaptive grid* programs, such as linear differential equation solvers[21]. In such a program, a grid is constructed and the function solved at each point in the

```
(agent sort-abst
       (ports inport outport)
       (on inport (lambda  (message)
                   (if (eq? message eof-object)
                       (send outport '())
                       (rewrite sort-body message)))))

(agent split
       (args seed)
       (ports in hi seedport lo)
       (send seedport seed)
       (loop in (lambda (in)
                  (if (not (eq? in eof-object))
                      (begin
                        (if (< in seed)
                            (send lo in)
                            (send hi in)))
                      (begin
                        (send hi eof-object)
                        (send lo eof-object)
                        (break))))))

(agent join
       (ports hi seed lo out)
       (on-and (his seed lo)
               (lambda (hi seed lo)
                 (send out (append lo (list seed) hi)))))
```

Figure 3: Sort Example - Agent Code

grid. Grid points in each other's neighborhood then transmit their solutions to one another. If there is too large a discontinuity between results at any point, that point is rewritten to a finer grid and the process repeated. Solutions to such problems find natural expression in GARP.

6 Graphs and Abstractions

We can now summarize how CAGs help control network topologies. Rather than allowing agents to connect to other agents in arbitrary ways, the interconnections are taken care of by the CAG. We see several advantages to this approach. First, it forces grouping of agents (via productions) that will cooperate together to perform some aspect of the computation. It is easy to see which agents will work together; one just has to look at the CAG. Second, interconnection topologies are determined at the level of the CAG, not at the level of individual agents. This means that setting up topologies becomes the province of the designer rather than of the programmers implementing the agents, as good software engineering practice dictates.

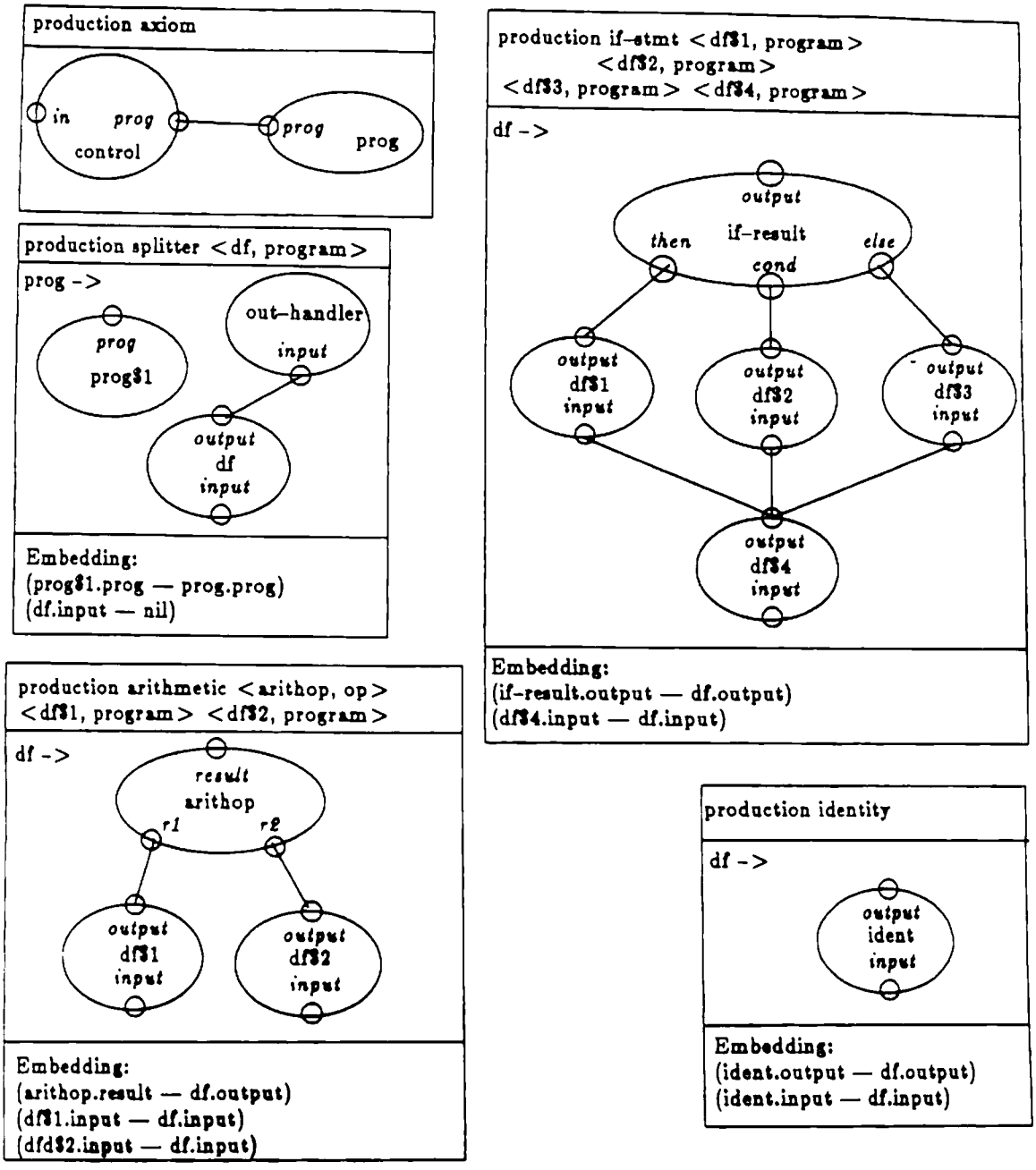


Figure 4: Dataflow Example - Graph Grammar

7 Related Work

There is a large body of literature on graph grammars (see, for example, [7]). Some researchers have developed very powerful formalisms where graphs can be rewritten to graphs rather than just rewriting vertices [8] [17]. This work is quite attractive on the surface, but would be almost impossible to implement: identifying the graph to be rewritten is NP-hard, and it is not clear how to synchronize the mutual rewriting of the vertices in the graphs. The primary focus of these researchers has been on theoretical issues such as the confluence of various classes of graph grammars and the hardness of the recognizability problem. We have instead based CAGs on a more limited form of graph grammar, Node Label Controlled (NLC) grammars [11]. The basic difference between CAG and NLC grammars is each CAG production has its own embedding rule. Our "semi-confluence" theorem does not, to our knowledge, appear in the literature. GARP can be viewed as an extension of NLC grammar research into a more practical domain.

Kahn and MacQueen [12] have investigated a parallel programming model in which individual processes are replaced by networks; while our work is similar, the major difference is that we have a formal way of modelling the network topologies that are created.

Degano and Montanari [5] have used a graph grammar formalism similar to CAG as the vehicle for modeling distributed systems. Although their work differs from ours in several respects—a more restricted model of embedding is used, there is no model of communication among processes, graphs in their formalism carry history information, and the grammars are used to model programs rather than as a programming formalism in their own right—it is still an interesting complement to our work, and we believe that many of their results will be transferable.

GARP is most similar to Actors[4] [3] [9]. An important difference is that in GARP communications patterns are defined in the grammar, whereas in actors they are set up by passing of addresses among Actors. We believe that this lack of structure is potentially dangerous, as it relies on the goodwill and cooperation of the programmers building the system. As long as the programmers continue to cooperate successfully, the system will work; but the smallest error in propagation of Actor addresses could lead to chaos. Experience with large software systems written in sequential programming languages strongly suggests that lack of suitable structuring constructs for the network will cause serious software engineering problems. An attempt to address this problem using *receptionists* allows the programmer to break up the Actors into groups by convention only; a mischevious programmer may still break the system by passing "internal" Actor addresses out to other Actors. In GARP this cannot happen.

[4]Space dictates that we assume the reader is familiar with Actor systems

The distinction between the process spawning supported by Actors and by GARP is analogous to the replacement of conditional and unconditional branches in sequential programming languages with structured control constructs. The distinction between the communication patterns is analogous to the distinction between dynamic and lexical scoping.

Two other ways of describing parallel networks—CCS [16] and Petri Nets [18]—are also related to our work. With CCS we share the concept of ports and the idea of a network of processes; however, we use asynchronous communication where CCS is synchronous and needs no notion of global time. It also seems that the application of CCS is limited to fixed topology networks. Petri nets use asynchronous communication, but are also limited to fixed topology.

There are several other approaches to concurrent programming that we have cited in the text: Ada focuses on providing a good language model for a process, and all but ignores interprocess topology issues; Cantor is interested in parallel object-oriented programming and gives the same support for topology control as does Actor Systems; and Argus focuses on issues of atomicity and robustness. These issues are orthagonal to those addressed in this paper.

8 Conclusions

MIMD computer systems make inevitable the development of large parallel programs. At present there are no adaquate ways to specify the interconnections among processes in these programs. We believe that this will lead to a situation in which programs can generate completely arbitrary process topologies. Such programs will be difficult to debug, verify, or maintain. This problem is analogous to the "goto problem" of the 1960's, and we propose an analogous solution: rather than being able to construct arbitrary networks, abstractions should be imposed that control network structure. However, unlike the "goto problem", we do not believe that it will be possible to derive a set of standard form similar to the "if" and "do" forms used in sequential programming; rather, we believe that for each parallel program, the designer should identify a set of interconnection topology templates and use those as the abstractions for that program.

Graph grammars provide an excellent medium in which to encode these templates, and in the GARP system we have shown that a mechanical interpretation of a subclass of graph grammars - CAG grammars - does indeed allow the specification of interprocess connections and their automatic use in a parallel programming system.

Acknowledgements

Thanks to Roy Campbell and Steve Goering for frequent discussions on the GARP system and the theory underlying it, as well as their comments on earlier drafts of this paper.

References

[1] *Reference Manual for the Ada Programming Language.* Technical Report MIL-STD 1815, United States Department of Defense.

[2] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* M.I.T. Press, Cambridge, Mass., 1986.

[3] Gul Agha. Semantic considerations in the actor paradigm of concurrent computation. In A. W. Roscoe S. D. Brookes and G. Winskel, editors, *Seminar on Concurrency, LNCS 197*, pages 151–179, Springer-Verlag, New York, 1985.

[4] W. C. Athas and C. L. Seitz. *Cantor User Report.* Technical Report 5232:TR:86, California Institute of Technology, January 1987.

[5] Pierpaolo Degano and Ugo Montenari. A model for distributed systems based on graph rewriting. *J. ACM*, 34(2):411–449, April 1987.

[6] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[7] Hartmut Ehrig, Manfred Nagl, and Grsegorz Rozenberg (eds). *Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science 153.* Springer-Verlag, 1984.

[8] Hartmut Erhig. Introduction to the algebraic theory of graph grammars. In Hartmut Erhig Volker Claus and Grsegorz Rozenberg, editors, *Graph Grammars and their Application to Computer Science and Biology*, pages 1–69, Springer-Verlag, Heidelberg, 1979.

[9] C. Hewitt, T. Reinhart, G. Agha, and G Attardi. Linguistic support of receptionists for shared resources. In A. W. Roscoe S. D. Brookes and G. Winskel, editors, *Seminar on Concurrency, LNCS 197*, pages 151–179, Springer-Verlag, New York, 1985.

[10] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[11] Dirk Janssens and Grzegorz Rozenberg. Graph grammars with node-label control and rewriting. In Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, editors, *Proceedings of the second International Workshop on Graph Grammars and their Application to Computer Science, LNCS 153*, pages 186–205, Springer-Verlag, 1982.

[12] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 77*, pages 993–998, Academic Press, 1978.

[13] Gail E. Kaiser and David Garlan. Melding data flow and object-oriented programming. In *Conference on Object Oriented Programming Systems, Languages, and Applications*, Kissimmee, FL, October 1987.

[14] Barbara Liskov and Robert Scheifler. Guardians and actions: linguistic support for robust, distributed programs. *ACM TOPLAS*, 5(3):381–404, July 1983.

[15] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming, Volume 1.* Addison-Wesley, Reading, Mass., 1985.

[16] R. Milner. A calculus of communicating systems. In *Lecture Notes in Computer Science volume 92*, Springer-Verlag, Berlin, 1980.

[17] Manfred Nagl. A tutorial and bibliographical survey on graph grammars. In Hartmut Erhig Volker Claus and Grzegorz Rozenberg, editors, *Graph Grammars and their Application to Computer Science and Biology*, pages 70–126, Springer-Verlag, Heidelberg, 1979.

[18] C. A. Petri. Concurrency. In *Net Theory and Applications, LNCS 84*, Springer-Verlag, Berlin, 1980.

[19] J. Rees and W. Clinger (Editors). Revised (3) report on the algorithmic language scheme. *Sigplan Notices*, 21(12):37–79, December 1986.

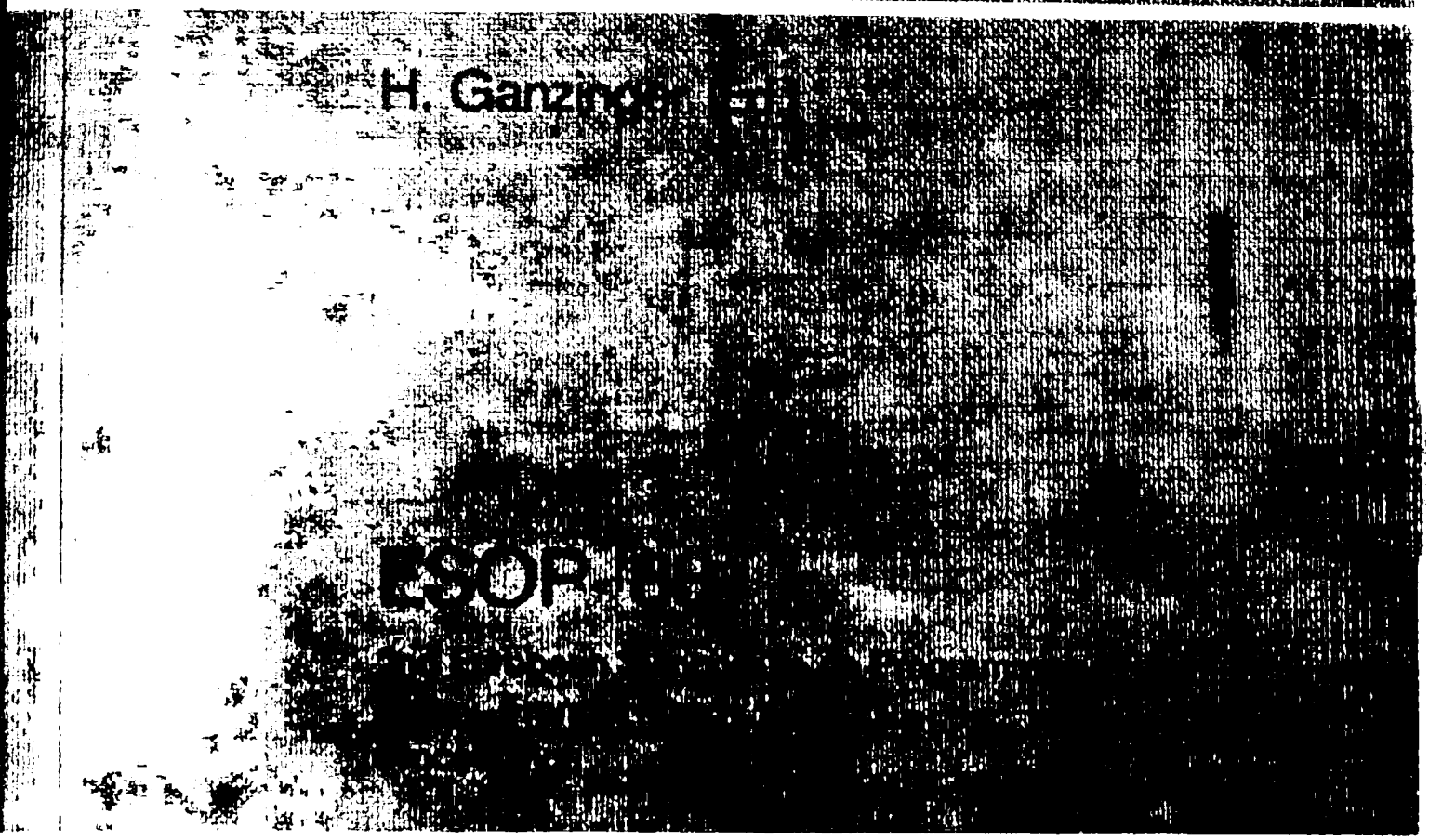
[20] Robert E. Strom and Shaula Yemini. The nil distributed systems programming language: a status report. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar On Concurrency, LNCS 197*, pages 512–523, Springer-Verlag, New York, 1985.

[21] J. F. Thompson, Z.U.A Warsi, and C. W. Mastin. *Numerical Grid Generation: Foundations and Applications*. North-Holland, New York, 1985.

[22] William M. Waite and Gerhard Goos. *Compiler Construction*. Springer-Verlag, New York, 1984.

Lecture Notes in Computer Science

300



H. Ganzinger (Ed.)

ESOP '88

Springer-Verlag

Garp: A Graphical Language for Concurrent Programming

Simon Kaplan

Roy Campbell Steven Goering

Joe Loyall Susan Hinrichs Brenda Jackels

University of Illinois

Department of Computer Science

Urbana, IL 61081

Gail Kaiser

Department of Computer Science

Columbia University

New York, NY 10027

Abstract

The advent of the large, inexpensive MIMD computer makes inevitable the development of large concurrent software systems. Such systems will often have *dynamic* topology: the number of processes and their interconnections will change. It is therefore imperative that we develop programming notations to simplify the specification of systems involving changing topologies. Such a notation must not only specify connections and name processes, but it must provide user-defined abstractions that facilitate the programming of complex topologies by structuring and simplification. This paper proposes a graph grammar based-approach to specifying changing topologies. We introduce the GARP system, a programming environment that implements this graph grammar approach and give solutions to example problems in which the topologies of concurrent systems dynamically change.

keywords: abstraction, actors, concurrency, distributed system, graph grammar, message passing, object-oriented system, parallel processing

1 Introduction

This paper proposes a method for specifying the topologies of dynamically changing processes and their communications in programs that describe massively concurrent systems. The proposal is based both on theory and on experiment. GARP [1]. is an example software programming system that we have built to demonstrate the proposal.

The problem which we study has the following characterization. In a concurrent system with dynamic topology, processes are frequently created and destroyed. As some processes are created and destroyed, other processes must update their communication links to accommodate the changes. Without a suitable programming notation to specify such changes, a system cannot easily be designed that is free from error. The problem can be compared to the unstructured use of goto statements in sequential programs. A notation can be used to aid the "structured programming" of dynamic concurrent systems in much the same way as `while` and `for` loop constructs were introduced to structure sequential programs. However, in at least one respect, the problem of structuring dynamic concurrent systems is more complicated than that of the goto problem. Because concurrent systems are nondeterministic, errors are often hard to reproduce. Thus, the activities of debugging and maintaining dynamic concurrent systems can be many times more difficult than the sequential analogy if a notation for structuring the programming of the system is not used.

The problems of programming dynamic concurrent systems is exemplified in the following adaptation of the five philosophers problem introduced by Dijkstra. In the original problem, each philosopher repeats a cycle of eating and thinking, where each activity lasts for a variable length of time. To eat, a philosopher must sit at a table and pick up two forks, a right fork and a left fork. (Picking up a fork is represented by performing a P operation on a semaphore.) There are five forks, each fork is both a left and a right fork for two separate philosphers. If the philosophers should all, by chance, sit down together and pick up a fork to their left (or right) concurrently, the result would be deadlock unless one or more of them can put down a fork. Should the philosophers be allowed to put down a raised fork, starvation may occur because all the philosophers may put down their forks

[1] "Garp" stands for *Graph Abstractions for concuRrent Processing.*

concurrently. A solution to the problem must prevent deadlock or the starvation of any of the philosophers.

In the dynamic adaptation of this problem, we allow the number of philosophers and forks to vary. A new philosopher must both bring a new fork to the table and be willing to share that fork with another philosopher. The philosophers already eating and thinking must adapt to the addition or removal of a philosopher. A departing philosopher may remove a shared fork from the table. Any solution to this problem requires a program that dynamically changes the binding of resources to processes. An "ideal" solution should permit the concurrent arrival and departure of philosphers and their forks while still preventing deadlock and starvation. A "structured" solution to this problem would be composed of two parts, a solution to the programming of the appropriate synchronization involved in sharing forks among philosphers and a solution to the programming of the changing topology of the philosophers and forks. GARP is a language that we have designed to express the solution to such problems clearly and precisely.

The method that specifies dynamic topologies in GARP separates the issues of programming agents (both processes and resources in GARP are called agents) from the problem of specifying the changing topology of the dynamic concurrent system. Each agent has a set of ports through which it communicates with other agents. These ports are abstractions for (sets of) other agents. A particular topology is specified by means of a graph that describes the possible communications (the edges of the graph) between the agents (the vertices of the graph) in an implementation independent manner. All communication among agents is asynchronous. The set of graphs that represent valid topologies of the concurrent system is described by means of a graph grammar. A valid change to the topology of the concurrent system is specified by the application of a production of the graph grammar to an agent of the system. That is, a change corresponds to a rewrite rule that modifies a process or resource of the underlying concurrent system so that it conforms to the graph created by applying the production.

A graph grammar is similar to a string grammar, except that the right hand side of the productions are graphs rather than trees. Each vertex in the productions is labeled by a

terminal or nonterminal symbol, representing an agent. The left hand side of a production is a nonterminal symbol. The rewriting action on graphs replaces a vertex labeled by a nonterminal with the body (right hand side) of a production for which that nonterminal is the goal symbol. This replacement involves the connection of the instantiation of the body to the vertices already in the graph (a process known as embedding); we restrict this embedding to be to vertices that were in the neighbourhood (connected by a path of length 1) of the vertex being rewritten.

The theory behind GARP is presented in [11]. This paper extends previous work on GARP by enhancing the agent language with an object-oriented style in order to reduce the effort needed to write programs. (The earlier work used a CSP-like [7] approach to reading from and writing on ports.) We believe the issues of programming agents and programming topologies are orthogonal; GARP programs agents independently from the topology of the system (the communications through the ports.) The use of a graph grammar based notation provides a method of declaring the specification of a dynamic concurrent system in a manner that can be readily analyzed. Each production of the graph grammar represents an abstraction of the ways in which the system can change; it describes the manner in which an agent and its communications can be transformed.

The GARP system provides a structured way of building concurrent systems. The graph grammar simultaneously acts as a design document, a process interconnection language and an abstraction mechanism.

We should note that few languages that currently support concurrent programming, such as Ada [1], CSP [7], NIL [16] or Argus [12] provide any high-level specification of topologies. Most provide a model for processes and primitives for communication while ignoring structural interconnection issues. Even Actor Systems [2], in which topologies can be flexible, can only control topologies by passing around addresses of processes, and these primitives are analogous to pointers or goto statements.

In the body of the paper we take a bottom-up approach to the motivation and introduction of the GARP system. Section 2 discusses agents, and gives some examples. Section 3 overviews graph grammars and discusses their application to topology control. Section 4

illustrates the use of the GARP approach with some examples. Section 5 discusses the implementation and section 6 discusses related work.

2 Agents

All computation in GARP is carried out by agents. An agent

- is an independent entity that computes asynchronously from all other entities.
- communicates with other agents by sending messages. This message sending also is asynchronous.
- implements a set of *methods*, which define the messages to which the agent can respond.
- has a set of *ports* on which messages for other agents are written, and through which the replies to those messages are returned. Each port can be connected to a set of agents, and a sophisticated selection mechanism is provided for choosing subsets of the agents connected to a port.

We will define agents more completely in the body of the section.

Agents are most similar to Actors; the major differences are (1) that agents do not explicitly know or deal with addresses of other actors; and (2) agents have a port concept that acts as an abstraction of other actors. The question of how topology control is achieved in GARP is deferred to the following section.

An agent is defined using the syntax in figure 1. An agent a is a named, parameterized object with a set of ports $\mathcal{P}_a$, a set of methods $\mathcal{M}_a$, internal definitions and a list of initialization expressions. The methods define the set of messages to which the agent can respond. The ports act as abstractions of (sets of) other agents with which this agent can communicate. The internal definitions allow the declaration of local storage ("instance variables" in object-oriented jargon) and procedures. The initialization expressions are executed when an instantiation of the agent is created. The parameters to the agent allow the passing in of state information when an instantiation is created.

```
(agent  name
   (ports  name ...)
   (args  name ...)

   (define ...)            ;; internal definitions here

   (method  name lambda-expression)
   (method  name lambda-expression) ...

    expression ... )
```

Figure 1: Syntax for Agents

In order to ease the implementation of the GARP language we built the implementation on top of Scheme [15]. GARP therefore inherits the features of Scheme including static scoping and the ability to pass and return procedures to and from other procedures as values; the `lambda-expressions` in the definition of an agent are just Scheme lambda-expressions, and are first class values.

The set of methods $\mathcal{M}_a$ defined in an agent specifies the messages to which the agent can respond, and the expected arity (number of arguments) of each message. For simplicity in this paper, we assume that all methods are explicitly given in the agent definition. In practice it is possible to add an inheritance hierarchy to the language to permit code sharing and the development of common interfaces among groups of functionally similar agents. In object-oriented terms, the graph constructed during a GARP program execution is an *instance* interconnection structure; adding inheritance is therefore an orthagonal issue which we do not discuss further here.

Each port $p \in \mathcal{P}_a$ in an agent a is typed by a set of message names, denoted $\mathcal{T}_{p_a}$ or $\mathcal{T}_p$ if the specific agent is irrelevent or determinable from the context. This typing imposes the following restriction on a legal GARP graph: when an agent a writes a message m on a port p to which are connected a set of agents $\mathcal{A}$ such that $m \in \mathcal{T}_{p_a}$, the predicate $\forall a \in \mathcal{A} : m \in \mathcal{M}_a$ must hold, *i.e* all the agents must be able to respond to message m.

When an agent is instantiated, each port is connected to a (possibly empty) set of agents. When a message is written on a port, it is forwarded to the set of agents connected to the port, and the results of the processing of the message at the remote agents are passed back to the sending agent. Often the agent wishes to communicate with some subset of the agents connected to the port; to support this a flexible selection mechanism is provided, which we will describe below. Each message sent is encoded with the address of the sender, to facilitate replies.

Ports can thus be thought of as acting as abstractions of (sets of) agents. From the viewpoint of the programmer writing the code for an agent, the ports themselves are active objects; the interfacing to the system, naming of remote agents, and identifying their locations are all hidden from the programmer.

Messages are sent out of a port using the form

```
(portexp message arg ...)
```

where the portexp is either a port name, in which case it names all the agents connected to the port, or it is a `select` expression as defined below, in which case the subset of the agents connected to the port which satisfy the selection criteria are named by the portexp. The message indicates which message is being sent and the `arg ...` are the arguments to the method responding to the message. The portexp is evaluated, yielding a list of agents. These are then each sent the message and arguments. The result of the message send is a list of replies, one for each agent identified by the portexp. Each reply is actually a pair; the reply and the address of the agent that generated it. The latter information is for use by the `select` operator explained below, and is not directly accessible by the programmer.

The `select` operator allows the programmer to select a subset of the agents connected to a port without knowing their addresses, how many agents there are, what their states are, or what states they were in or message replies they returned when previous messages were sent out of the port. This ability to work at such a high level is very important as the network topology is dynamic. We cannot expect knowledge about agents connected to a port to remain current for any longer than the evaluation of a portexp. Instead we always select by sending a message out the port using the first form, and then apply some

relation to the replies to select the required subset. The general form is

```
(select predicate agentlist arg ...)
```

The `predicate` is any predicate, the `agentlist` is a list of agents connected to the port, obtained as the result of the message sending form above, or from a previous `select`. The `arg ...` are once again additional arguments, this time to the `predicate`. The result part of each pair in the `agentlist` is submitted to the `predicate` along with the arguments. If the predicate evaluates to true, the reply pair for that agent is returned, otherwise a "null" pair is returned. The list of all agents that satisfy the predicate is the result of the `select` operator. That list can then be used immediately in a message send, or passed through another selection. `Select` can be thought of as a specialized form of the Scheme procedure `map`.

The following example illustrates the use of selection. We assume that this code is written for a user agent, which wishes to send a file to an agent dedicated to processing text files using `troff` or TEX. The agent has a `textprocessor` port which communicates with *all* such agents, but only a subset support each type of text processor. The following selection chooses the subset that handles TEX jobs:

```
(select eq?  (textprocessor 'type?)  'latex)
```

This sends the `type?` message to every agent connected to the `textprocessor` port, and then chooses the subset whose type was equal to `latex`. We can then use this selection in a more complex one that chooses the machine with lowest load:

```
((select min ((select eq?  (textprocessor 'type?)  'latex) 'load?))
                        'process textfile)
```

This selects the TEX processors as before, then sends those the `load?` message, and then selects the least loaded processor as the recipient of the file. That processor is then sent the `process` message, along with the file to be processed.

This approach allows the programmer to work at a level independent of knowledge about the network or the number of text processing servers to which his agent is connected at any time.

There are two other forms used occasionally when working with messages. The first of these, (`break expression`) is designed for use in case where the reply to a message is not needed. The `expression` is a message send; and rather than wait on a reply, the `break` causes execution to move to the next expression in the agent (technically, the continuation of the `break`). The value of a `break` is undefined.

The second form is (`reply expression`) and is useful in case where a method wishes to send a reply and then continue with some processing. The `expression` is evaluated and its result sent as the result of the method. Any other `reply` in the method before it terminates is evaluated (for side-effects) but its result is not sent to the originator of the message that triggered the method's execution.

As an example of an agent, consider the agents that implement forks and philosophers for the dining philosophers problem, shown in figures 2 and 3. Note how we assume that each fork port is connected to exactly one fork. Forks have no ports; this is because they initiate no communications. We have not yet indicated how the agents are instantiated, or how they are wired together to produce a working concurrent system; this is the subject of the next section. Later we will use these agents in a solution to the dynamic dining philosopher problem, in which the number of philosophers (and forks) can change.

3 Using Graph Grammars to Interconnect Agents

Thus far we have introduced and explained our concept of an agent. Agents assume that there is an underlying network into which they have been spliced in some way, which takes care of naming issues for them. The purpose of this section of the paper is to introduce graph grammars and explain how they are used to achieve this. Section 3.1 briefly reviews formal graph grammar issues, and section 3.2 discusses the integration of grammars and agents. Examples are deferred to section 4.

3.1 Graph Grammars

Graph grammars are similar in structure to string grammars. There is an alphabet of symbols, divided into three (disjoint) sets called the terminals, nonterminals and portsymbols.

Productions have a nonterminal symbol as the goal (the same nonterminal may be the goal of many productions), and the right-hand side of the production has two parts: a graph (called the bodygraph) and an embedding rule. Each vertex in the bodygraph is labeled by a terminal or nonterminal symbol, and has associated with it a set of portsymbols. Any portsymbol may be associated with many terminals or nonterminals.

Edges in the graphs are directed, and always go from a port associated with a vertex to some vertex (possibly the same vertex). This captures the notion that a port on an agent is connected to some other agent.

The rewriting action on a graph (the host graph) is the *replacement* of a vertex labeled with a nonterminal by the bodygraph of a production for which that nonterminal is the goal, and the *embedding* of the bodygraph into the host graph. This embedding process involves placing edges from ports associated with vertices in the bodygraph to vertices in the host graph, or from ports associated with vertices in the host graph to vertices in the bodygraph. The embedding process is restricted so that when a vertex v is rewritten, only vertices that are in the *neighborhood* of v–those connected to v by a path of unit length–can be connected to the vertices in the bodygraph that replaces v.

Because we use these graph grammars as an abstraction construct for concurrent programming, we call them concurrent abstraction grammars (CAGs).

Each symbol in the alphabet of terminals and nonterminals has associated with it a set of symbols called *portsymbols*. The same portsymbol may be associated with several terminals or nonterminals. We denote terminals and nonterminals by uppercase characters $X, Y, \cdots$ and portnames by Greek characters $\alpha, \beta, \cdots$. Vertices are denoted $v, w, \cdots$ and the symbol labeling a vertex v is identified by Lab_v. PS_X denotes the set of portsymbols associated with the (terminal or nonterminal) symbol X.

For any graph G, let V_G denote the vertices in G and E_G the edges of G. Each vertex v can be qualified by the portsymbols in PS_{Lab_v} to form a *port-identifier*. Edges are denoted by pairs, the first element of which is a port identifier and the second is a vertex, for example $(v.\alpha, w)$. This indicates an edge from port α on vertex v to vertex w. For any vertex v in a graph G, the neighborhood of v, $\mathcal{N}_v$, is $\{w \mid (v, w) \in E_G\}$.

Definition 1 *A concurrent abstraction graph grammar is a tuple* $CAG = (N, T, S, P, Z)$, *where* N *is a finite set of symbols called the nonterminals of the grammar,* T *is a finite set of symbols called the terminals of the grammar and* S *is a finite set of symbols called the portsymbols of the grammar such that* $T \cap N = N \cap S = T \cap S = \emptyset$; P *is a set of productions, where productions are defined in definition 2 below; and* Z *is a unique distinguished nonterminal known as the axiom of the grammar.*

The axiom Z is the goal of exactly one production and may not appear in any bodygraph. This requirement is not a restriction in practice as one can always augment a grammar with a distinguished production that satisfies this requirement.

Definition 2 *A production in a CAG is defined as:* $p : L_p \rightarrow B_p, F_p$ *where* p *is a unique label;* $L_p \in N$ *is called the goal of the production;* B_p *is an arbitrary graph (called the bodygraph of the production), where each vertex labeled by an element of* $T \cup N$; *and* F_p *is the embedding rule of the production: a set of pairs each of which has one of the following forms:* $(X.\alpha, L_p.\gamma)$, $(Y.\beta, X)$ *or* $(X.\alpha, Y)$, *where* X *labels a vertex in* $B_p, Y \in \mathcal{N}_{L_p}$ *or* Y *is the special wildcard character "*" or* Y *is the special symbol %,* $\alpha \in PS_X, \beta \in PS_Y, \gamma \in PS_{L_p}$. *These terms are explained below.*

The same symbol may appear several times in a bodygraph; this is resolved by subscripting the symbol with an index value to allow them to be distinguished [17].

Definition 3 *The rewriting (or refinement) of a vertex* v *in a graph* G *constructed from a CAG by a production* p *for which* Lab_v *is the goal is performed in the following steps:*

- *The neighborhood* $\mathcal{N}_v$ *is identified.*
- *The vertex* v *and all edges incident on it are removed from* G.
- *The bodygraph* B_p *is instantiated to form a daughter-graph, which is inserted into* G.
- *The daughter graph is embedded as follows. For each pair in* F_p *of the form* $(X.\alpha, Y)$ *an edge is placed from the* α *port of each vertex labeled by symbol* X *in the daughter-graph to any agent bound to symbol* Y *in* $\mathcal{N}_v$. *For each pair of the form* $(Y.\beta, X)$ *an*

edge is placed from the β port of each vertex bound to label Y in $\mathcal{N}_v$ to each vertex labeled X in the daughter graph. In case where Y is the special wildcard character "" then an edge is placed from (to) every symbol in the neighbourhood that has a portsymbol β. In case where Y is the special symbol % an edge is placed from (to) the vertex in the neighbourhood which sent the message which triggered the rewrite. In both cases, if the edge is placed from the vertex in the neighbourhood, it comes from the port labeled β.*

The remaining form in F_p uses the edges incident on v before the rewriting to place new edges. This is useful in case where the writer of the grammar does not wish to use the neighbour symbols explicitly (such as when a production is to be used in many different contexts). For the form $(X.\alpha, L_p.\gamma)$ an edge is placed from *the α port of each vertex labeled by X in the daughter-graph to the agents that were connected to the γ port of v before the rewriting began.*

For further definitions, and a proof that CAGs have a limited confluence property (any two vertices not connected by a path of unit length may be rewritten in parallel) see [11].

3.2 Relating Graph Grammars and Agents

We can now address the issue of relating agents and CAGs, and describe how to execute the two as a concurrent system. A GARP program has two parts: a CAG and code for each agent. Vertices in the graph grammar represent agents. Each agent name is bound to either a terminal or nonterminal symbol of the grammar. For a terminal or nonterminal symbol X, the ports defined by the agent must match PS_X. We extend the reportoire of the agents to include a **rewrite** operation with form:

(rewrite name exp ...)

where name is the label of a production that has the name of the agent about to be rewritten as goal and the exp ... are parameters to the production. The interpretation of this operation is the definition of rewriting given in section 3.1. The **rewrite** action must be the agent's last, because the model of rewriting requires that the agent be replaced by

the agents in the bodygraph of the production used in the rewriting. We will write of nonterminal and terminal agents, depending on the symbol to which the agent is bound in the grammar

We extend the production labels of graph grammars to have a list of formal parameters. Each element of the list is a pair (`agent`, `parameter`), which identifies the agent in the bodygraph of the production to which the parameter must be passed, and the specific formal parameter for that agent that should be used. When rewriting, the agents specified in the parameter list are passed the appropriate actual parameter when they are created. This ability to pass arguments from an agent to the agents that replace it provides a way to pass the state of the agent to its replacements. This feature is not unique to our agent system and can be found in Actors and Cantor[4].

The graph grammar productions give us a way of determining (in polynomial time) exactly which agents might potentially be connected to some port p; this allows typechecking of the ports to ensure that all these potentially connected agents support the correct set of methods.

With this machinery we can define the way that computation is achieved in GARP. When a GARP program begins execution, all the agents in the bodygraph of the production with axiom as the goal are instantiated (instantiation consists of creating an instance of the agent and starting its execution), and ports are connected as indicated by the edges in the bodygraph. When a nonterminal agent chooses to rewrite itself, it executes a `rewrite` operation, which modifies the graph by removing the agent and all incident edges, instantiates the agents in the bodygraph of the production being used in the rewrite and connects these new agents into the rest of the graph using the embedding rule. Note that connections are made to specific ports; in this way the "naming" of agents in the graph changes dynamically.

Rewriting is an atomic action within a neighbourhood, so agents must enter into a dialog with their neighbours to indicate that a rewrite will be beginning. If any edge to be removed in the first stage of the rewrite is being used in a `select` operation, then the rewrite must wait until the selection is complete or the edge is removed from contention in

the selection process.

When messages are sent to an agent, they are queued in an input buffer. An agent repeatedly nondeterministically selects a message from the buffer and applies the appropriate method (or returns an error if no appropriate method exists). When an agent is rewritten, any messages waiting in the input buffer are distributed according to the following scheme: For each message, determine its source (must be a neighbour of the agent being rewritten) and then determine the new agents to which the port of the source through which the message was sent are being connected. The message is then placed in the input buffer of each of these agents. Thus if an agent a was connected to some agent b through some port p, and b rewrites itself, any messages from a to b that have not yet been processed will be duplicated as many times as new edges are placed from p to new agents. The converse situation (having to deal with replies coming back to an agent that has been rewritten) cannot occur; An agent cannot rewrite itself while waiting for a reply to a message, because it is still in the message-send expression. It is possible that an agent can send a message and rewrite itself in case that the `break` operator is used, but in this case the reply is irrelevent.

4 Examples

We now illustrate GARP with several examples. We first present two solutions to the dining philosophers problem. The first is the traditional problem with a fixed number of philosophers. The second allows philosophers to join the table at random times. The second example illustrates a variable-density curve plotting algorithm. In this problem we wish to plot a number of points representing a curve, with the proviso that the more rapid the rate of change of the curve, the greater the number of points that should be plotted.

We look first at two versions of the dining philosophers problem, using the agents defined in section 2. The first is the traditional problem with a fixed number of philosophers. The graph grammar is trivial in this case (only one production is needed). Nonetheless the example is worthwhile as a gentle introduction to the GARP approach and as an indication of how easy it is to model traditional static topologies in GARP (including such things as

Petri nets [14], although we give no illustration of this). The second example considers a dynamic dining philosophers problem, in which the number of philosophers can change. This illustrates the ease in which dynamic topologies can be specified in GARP, with no need to change any of the agent code.

The first example is to be found in figure 4. The single production is instantiated, which takes the philosopher and fork code given above and creates as many copies as there are vertices with the appropriate labels, and generates instances, making the binding of fork ports in philosopher agents to the appropriate fork agents. From then on the system runs independently.

The extension of this example to one in which there is a static topology but the number of philosophers can be fixed at instantiation time is simple but space does not permit its consideration.

The second example allows the dynamic introduction and removal of philosophers. We focus mainly on philosopher introduction. A philosopher who wishes to join the group enters, bringing with him one more fork. He then attempts to join the table. This involves taking an existing philosophers fork connections, breaking one and inserting the new philosopher and fork. We do not allow the new philosopher to be so rude as to grab a fork which is currently in use; instead the new arrival must seat himself between a philosopher and a fork that is not in use by that philosopher.

To implement this we introduce a new nonterminal, called `maitre-d`. The maitre-d sits on the edge between a philosopher and a fork, and monitors the fork status. If a new philosopher enters and asks the maitre-d to be seated between the old philosopher and his fork, the maitre-d checks the fork status and if not in use, rewrites itself to a new philosopher and fork, together with appropriate new connections and new maitre-d nodes.

All this is illustrated in figure 5. Initially, for simplicity we assume we have two philosophers. The new agent `maitre-d` has its code shown in figure 4. The agent recieves all the messages that a `fork` agent could recieve. It monitors the flow of messages between the philosopher and fork to which it is connected, and maintains internally a status flag. It also recieves a new message, `add-new-philosopher`. On reciept of this message it returns

'OK or 'BUSY. In the former case it proceeds to rewrite itself to add a new philosopher to the system.

It is also possible to *remove* a philosopher and his fork from the network. This involves recording a refinement history of the network, and updating it with refinement and embedding information each time a rewrite occurs. Then, when an "unrewrite" is requested, this history information can be used to restructure the network appropriately. Space does not permit us to go into details; the data structures and algorithms to do this unrewriting are described in [10] (albeit in a slightly different context).

We look now at the example of variable-density curve plotting. In this example, we calculate the y-values of a function f given a set of x-values. Then we compare each pair of y-values, and if they differ more than a specified amount, we compute the function at some set of intermediate points. We do this repeatedly until all y-values are within a given tolerance of each other. The result of this is that where the curve is relatively flat, fewer points are computed; but as the curve becomes steeper, so more points are computed to define the curve more exactly. The grammar is given in figure 7, and the agent code in figure 8. Initially, we start with three points, the beginning, end and middle of the curve, as shown in the first production in the figure. The second production is used to add more points if needed.

5 Implementation

GARP is implemented using Scheme and X windows. It has been run on Suns, RTs, Vaxes and HP workstations. The working system currently consists of two parts:

- A graphical programming environment for creating grammars. This includes interactive type-checking of the grammars against a data dictionary of agent information. The environment is implemented in Scheme and X for portability.
- A simulator of the runtime behaviour of the system. This, too, is written in Scheme and has been in use for approximately one year.

A truly parallel implementation of the system is under development for the Encore Multimax. Once complete a distributed implementation on the Hypercube is planned.

6 Related Work

There is a large body of literature on graph grammars (see, for example, [5]). The primary focus of previous work has been on theoretical issues such as the confluence of various classes of graph grammars and the hardness of the recognizability problem. We have instead focused on practial application of the theory to concurrent programming. CAGs are based on a graph grammar formalism called Node Label Controlled (NLC) grammars [8]. The basic difference between CAG and NLC grammars is each CAG production has its own embedding rule.

An earlier paper on GARP [11] introduces a more primitive agent model (at approximately the level of CSP, but with dynamic topologies), and focuses more on theoretical issues. This paper differs from its predecessor in that it presents a more sophisticated, object oriented agent model in which ports are abstractions of sets of agents, introduces a model of selection on ports and shows how GARP allows an abstract naming model in the face of dynamically changing process networks.

Kahn and MacQueen [9] have investigated a parallel programming model in which individual processes are replaced by networks; while our work is similar, the major difference is that we have a formal way of modelling the network topologies that are created.

GARP is most similar to Actors[2] [3] [6]. An important difference is that in GARP communications patterns are defined in the grammar, whereas in actors they are set up by passing of addresses among Actors. We believe that this lack of structure is potentially dangerous, as it relies on the goodwill and cooperation of the programmers building the system. As long as the programmers continue to cooperate successfully, the system will work; but the smallest error in propagation of Actor addresses could lead to chaos. Experience with large software systems written in sequential programming languages strongly

[2]Space dictates that we assume the reader is familiar with Actor systems

suggests that lack of suitable structuring constructs for the network will cause serious software engineering problems. An attempt to address this problem using *receptionists* allows the programmer to break up the Actors into groups by convention only; a mischevious programmer may still break the system by passing "internal" Actor addresses out to other Actors. In GARP this cannot happen.

Two other ways of describing parallel networks – CCS [13] and Petri Nets [14] – are also related to our work. With CCS we share the concept of ports and the idea of a network of processes; however, we use asynchronous communication where CCS is synchronous and needs no notion of global time. It also seems that the application of CCS is limited to fixed topology networks. Petri nets use asynchronous communication, but are also limited to fixed topology.

There are several other approaches to concurrent programming: Ada [1] focuses on providing a good language model for a process, and all but ignores interprocess topology issues; Cantor is interested in parallel object-oriented programming and gives the same support for topology control as does Actor Systems; and Argus [12] focuses on issues of atomicity and robustness. These issues are orthagonal to those addressed in this paper.

7 Credits

GARP is principally the work of the first author, who also built the prototype. Campbell, Goering and Kaiser have contributed substantially to the design of the system. Loyall is responsible for the parallel implementation on the Multimax and Hinrichs and Jackels have built the X-based programming environment. Scheme is used as the implementation language throughout the project.

8 Conclusions

This paper has introduced the concurrent programming system GARP. GARP uses a multi-paradigm approach to solving the problem of building concurrent systems with dynamic interprocess topologies, in which issues of naming and topology control are described graph-

ically using a CAG graph grammar, and individual processes are specified using a textual programming language (our current base language is Scheme). This allows a separation of concerns when programming; the programmer can focus on the code for the process which he is developing, without being distracted by naming issues, and the designer of the system can specify all interconnections economically with a graph grammar, and not have to rely on the programmer implementing the correct addressing to make the naming in the system work properly.

References

[1] *Reference Manual for the Ada Programming Language.* Technical Report MIL-STD 1815, United States Department of Defense.

[2] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems.* M.I.T. Press, Cambridge, Mass., 1986.

[3] Gul Agha. Semantic considerations in the actor paradigm of concurrent computation. In A. W. Roscoe S. D. Brookes and G. Winskel, editors, *Seminar on Concurrency, LNCS 197*, pages 151–179, Springer-Verlag, New York, 1985.

[4] W. C. Athas and C. L. Seitz. *Cantor User Report.* Technical Report 5232:TR:86, California Institute of Technology, January 1987.

[5] Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg (eds). *Graph Grammars and their Application to Computer Science, Lecture Notes in Computer Science 153.* Springer-Verlag, 1984.

[6] C. Hewitt, T. Reinhart, G. Agha, and G Attardi. Linguistic support of receptionists for shared resources. In A. W. Roscoe S. D. Brookes and G. Winskel, editors, *Seminar on Concurrency, LNCS 197*, pages 151–179, Springer-Verlag, New York, 1985.

[7] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[8] Dirk Janssens and Grzegorz Rozenberg. Graph grammars with node-label control and rewriting. In Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, editors, *Proceedings of the second International Workshop on Graph Grammars and their Application to Computer Science, LNCS 153*, pages 186–205, Springer-Verlag, 1982.

[9] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 77*, pages 993–998, Academic Press, 1978.

[10] Simon M. Kaplan. *Incremental Attribute Evaluation on Graphs (Revised Version).* Technical Report UIUC-DCS-86-1309, University of Illinois at Urbana-Champaign, December 1986.

[11] Simon M. Kaplan and Gail E. Kaiser. Garp: graph abstractions for concurrent programming. In *ESOP '88*, Springer-Verlag, March 1988.

[12] Barbara Liskov and Robert Scheifler. Guardians and actions: linguistic support for robust, distributed programs. *ACM TOPLAS*, 5(3):381–404, July 1983.

[13] R. Milner. A calculus of communicating systems. In *Lecture Notes in Computer Science volume 92*, Springer-Verlag, Berlin, 1980.

[14] C. A. Petri. Concurrency. In *Net Theory and Applications, LNCS 84*, Springer-Verlag, Berlin, 1980.

[15] J. Rees and W. Clinger (Editors). Revised (3) report on the algorithmic language scheme. *Sigplan Notices*, 21(12):37–79, December 1986.

[16] Robert E. Strom and Shaula Yemini. The nil distributed systems programming language: a status report. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar On Concurrency, LNCS 197*, pages 512–523, Springer-Verlag, New York, 1985.

[17] William M. Waite and Gerhard Goos. *Compiler Construction.* Springer-Verlag, New York, 1984.

```
(agent fork
  (ports ())
  (args ())

  (define busy?
     (lambda ()
        code to determine
     fork status))

  (define concede-fork
     (lambda ()
        code to mark fork as
     in use and return 'OK reply))

  (define free-fork
     (lambda ()
       code to mark fork as free))

  (method P
      (lambda ()
          (if (busy?)
              'Fork-in-Use

(concede-fork))))

  (method V
      (lambda ()
        (free-fork)))


  (free-fork))
```

Figure 2: Fork Agent

```
(agent philosopher
   (ports leftfork rightfork)
   (args ())

   (define pick-up-forks
     (lambda (first second)

       (define retry
         (lambda ()
           (sleep-random-time)
           (pick-up-forks first second)))

       (if (eq? (first 'P) 'OK)
           (if (eq? (second 'P) 'OK)
               'Got-the-Forks
               (begin
                 (first 'V)
                 (retry)))
           (retry))))

   (define release-fork
       (lambda (fork)  (fork 'V)))

   (while #t (think)
             (pick-up-forks left right)
             (eat)
             (release-fork rightfork)
             (release-fork leftfork)))
```

Figure 3: Philosopher Agent

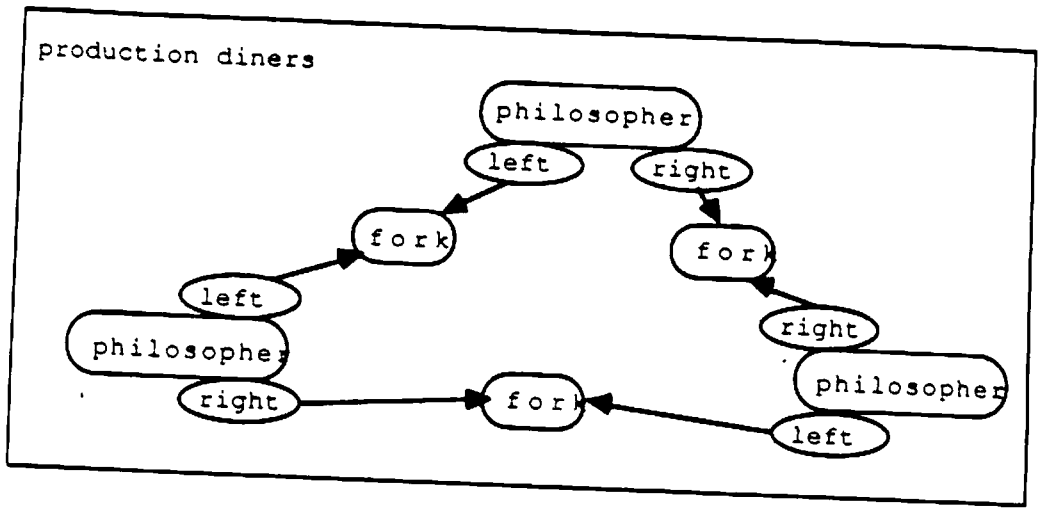


Figure 4: Static Dining Philosophers Grammar

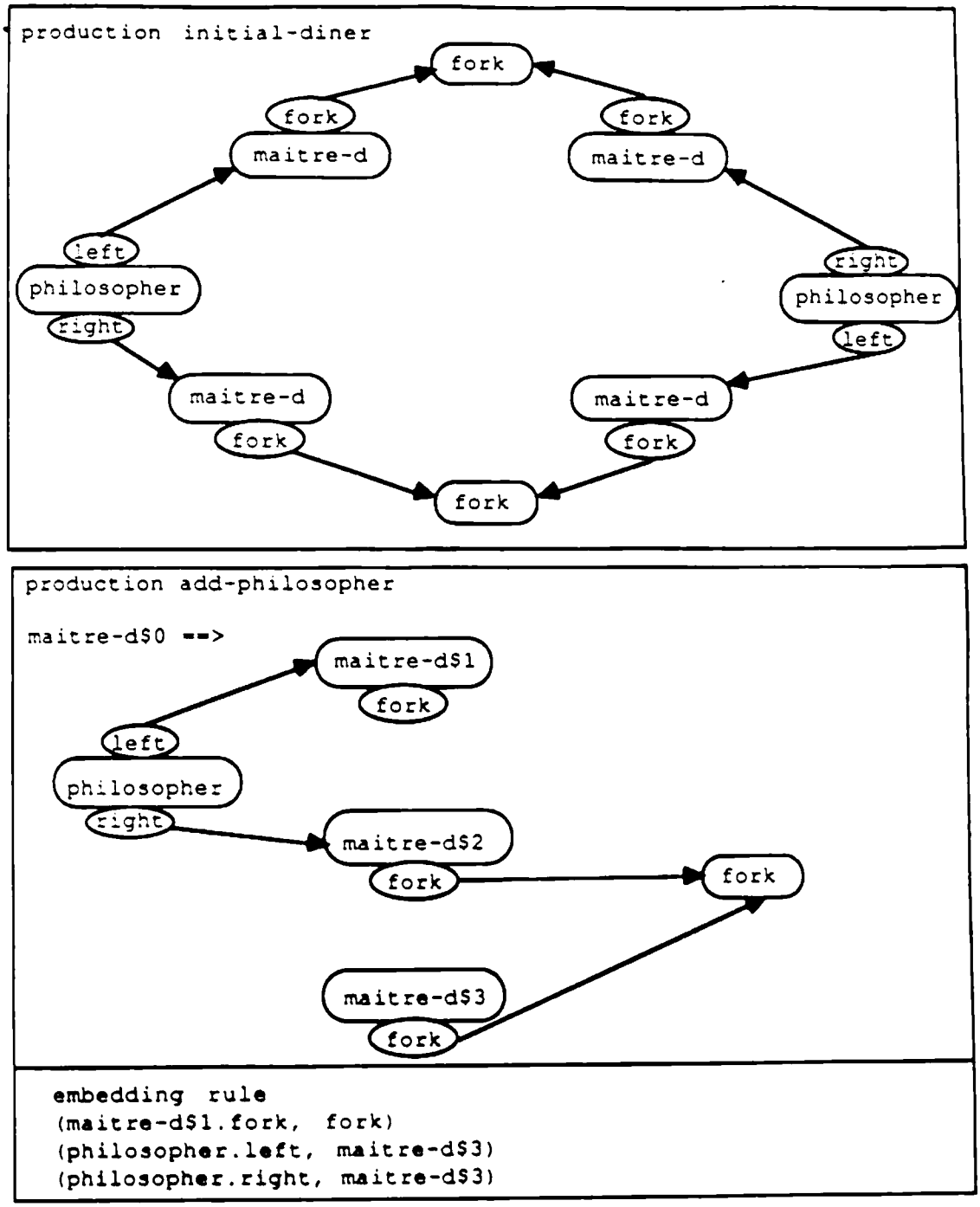


Figure 5: Dynamic Dining Philosophers Grammar

```
(agent maitre-d
  (ports fork)
  (args ())

   code for internal status manipulation omitted

  (method P
     (lambda ()
         (let ((temp (fork 'P)))
           (set-internal-status temp)
           temp)))

 (method V
    (lambda ()
       (reset-internal-status)
       (fork 'V)))

 (method add-new-philosopher
    (lambda ()
       (if (internal-status-ok?)
           (begin
              (reply 'OK)
              (rewrite add-philosopher))
           'BUSY))))
```

Figure 6: Code for Maitre-d nonterminal agent

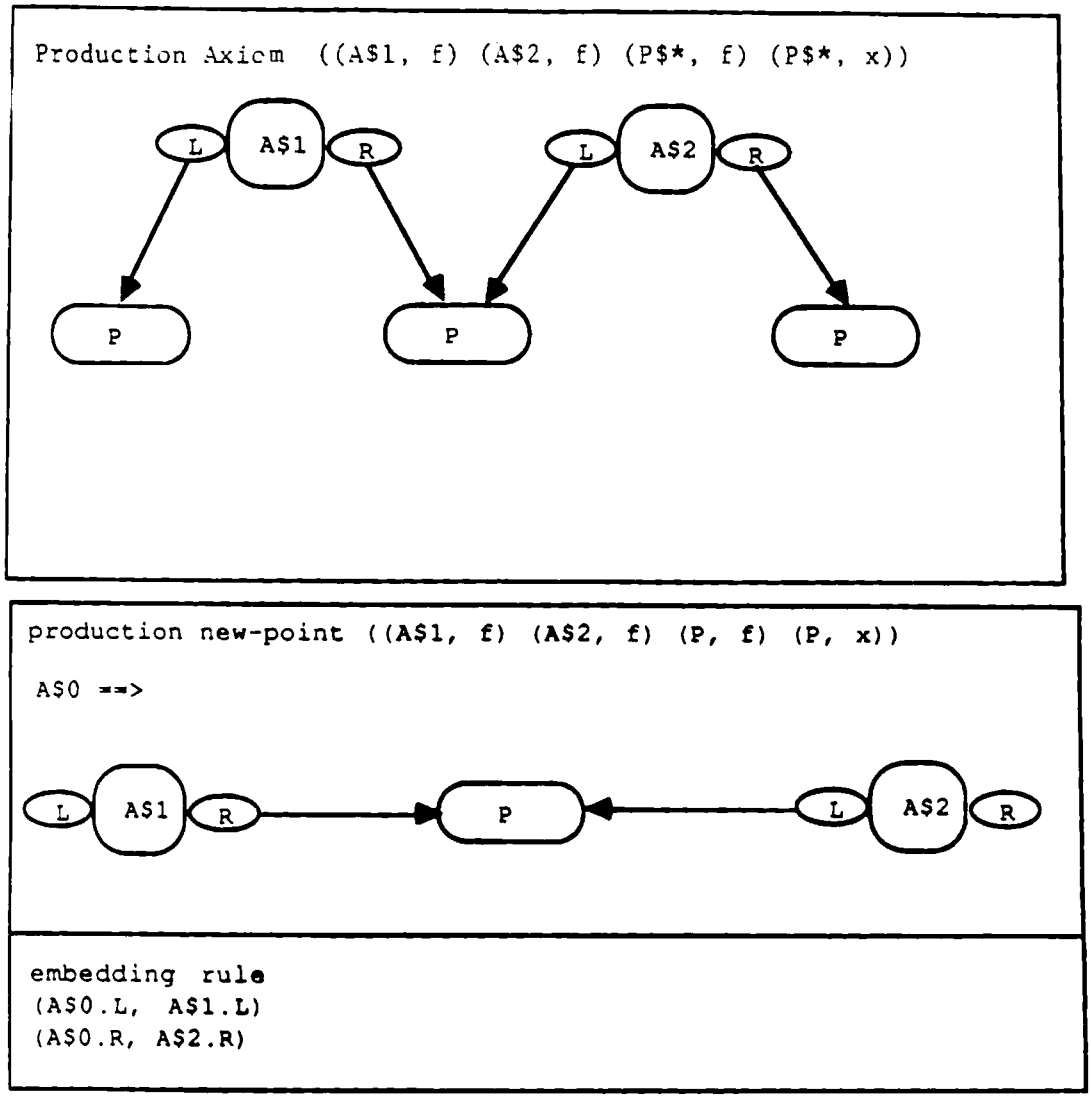


Figure 7: Variable Density Curve Grammar

```
(agent P
  (ports ())
  (args f x)

  (define val (f x))

  (method value
    (lambda ()
      (cons x val)))

(agent A
  (ports (L R))
  (args f)

  (define compare-results-and-rewrite
    (lambda (p1 p2)
      (if (not (close-enough? (cdr p1) (cdr p2)))
          (rewrite new-point f f f (average (car p1) (car p2))))))

  code for close-enough?, average not shown

  (compare-results-and-rewrite ((L 'value))
                               ((R 'value)))))
```

Figure 8: Variable Density Curve Agents