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 $\{w \mid w \in \mathcal{N}_v$ 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.

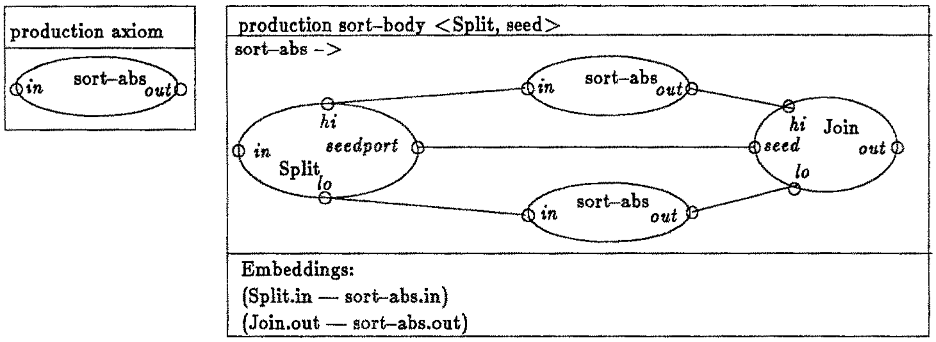

Figure 2: Sort Example – Graph Grammar

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 recognizes 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.

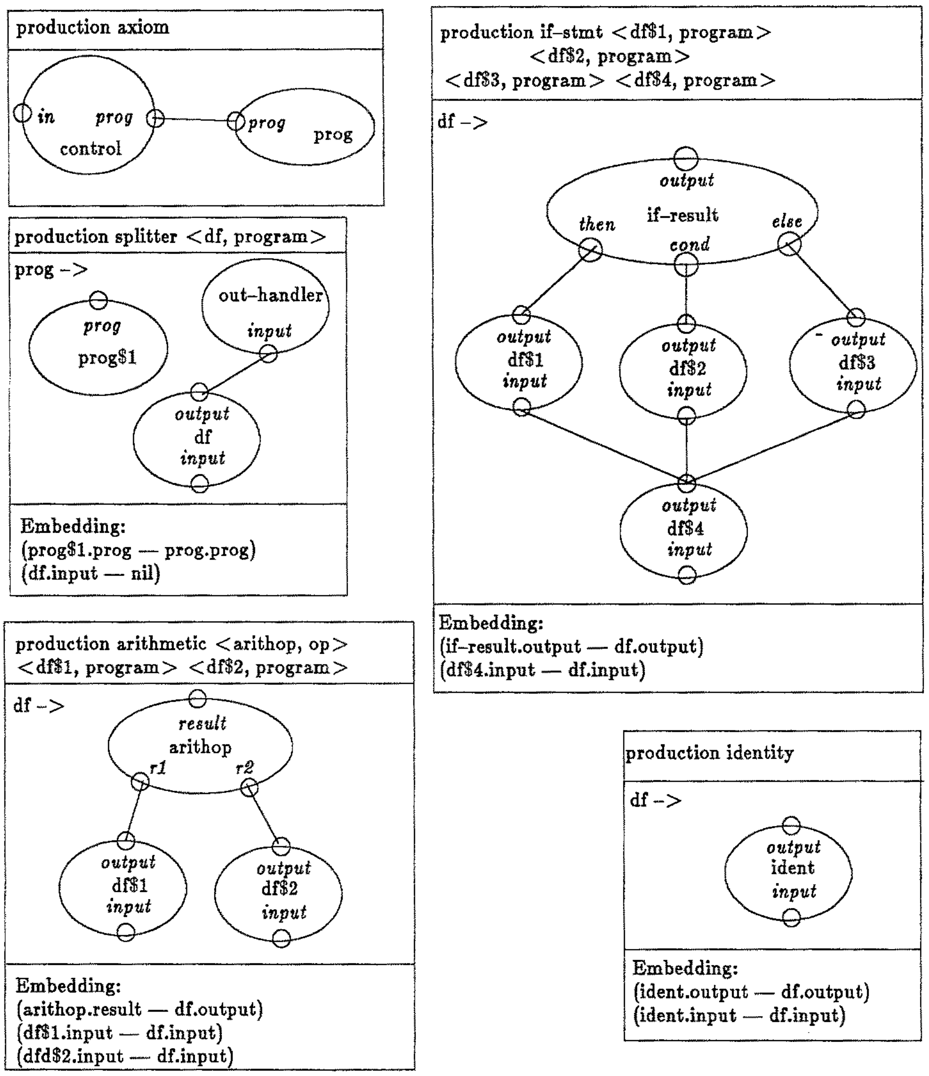

Figure 4: Dataflow Example - Graph Grammar

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 Grzegorz 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 Grzegorz 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.