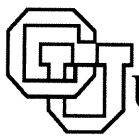
Collected Papers on Phred *

Adam L. Beguelin *
Gary J. Nutt **

CU-CS-511-91



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This work was done while this author was at the University of Colorado, supported by NSF Grant CCR-8802283. He is currently jointly appointed to the University of Tennessee Department of Computer Science and Oak Ridge National Laboratory Mathematical Sciences Division. His current address is Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301. His electronic mail address is adamb@cs.utk.edu.
** This author was supported by NSF Grant CCR-8802283. His electronic mail address is nutt@cs. colorado.edu.

Collected Papers on Phred

Adam L. Beguelin†
Gary J. Nutt‡

CU-CS-511-91 January, 1991

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430

---

† This work was done while this author was at the University of Colorado, supported by NSF Grant CCR-8802283. He is currently jointly appointed to the University of Tennessee Department of Computer Science and Oak Ridge National Laboratory Mathematical Sciences Division. His current address is Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301. His electronic mail address is adamb@cs.utk.edu.

‡This author was supported by NSF Grant CCR-8802283. His electronic mail address is nutt@cs.colorado.edu.

ABSTRACT

Collected Papers on Phred

Phred is a visual, parallel programming language designed to accommodate dynamic determinacy analysis as a program in the language is defined. Several papers and technical reports have been written describing the Phred language, supporting system, and examples. Beguelin's dissertation is the final word on most of the ideas in Phred. This collection includes three of the most significant papers about the work:

(1) A. L. Beguelin and G. J. Nutt, "A Visual Parallel Programming Language," October, 1990, submitted for publication.

(2) A. L. Beguelin, and G. J. Nutt, "Examples in Phred," presented at the *Fifth SIAM Conference on Parallel Processing for Scientific Computing*, Houston, Texas, March, 1991.

(3) A. L. Beguelin and G. J. Nutt, "A Tool and Language for Visual Distributed Programming," December, 1990, submitted for publication.

The first paper describes the programming language, the second provides examples of the usage of the language, and the third paper focuses on the properties and design of the system that supports Phred.

A Visual Parallel Programming Language *

Adam Beguelin †
Department of Computer Science
University of Tennessee, Knoxville

Gary Nutt ‡
Department of Computer Science
University of Colorado, Boulder

October 15, 1990

Abstract

Phred is a visual parallel programming language which can be statically analyzed for deterministic behavior. While nondeterminacy has often been employed as a technique for increasing the performance of parallel computations, it also implies that the program is nonfunctional. The inadvertent specification of nondeterministic (nonfunctional) computations is a common error in parallel programming. The ability to reveal such potential problems within a program may prove invaluable as increasing numbers of programmers construct parallel software. In this paper we describe Phred and indicate how it can be used to graphically specify parallel computations. We present the static analysis technique for detecting the possibility of nondeterminacy in Phred programs.

Index Terms: determinacy, graph grammars, graph models, graph parsing, parallel computing, parallel programming, programming environments, programming langauges, static analysis, visual programming.

*This work was supported by NSF Grant CCR-8802283.

†This work was done while this author was at the University of Colorado, Boulder.

‡Please direct correspondence to: Professor Gary Nutt, Department of Computer Science, University of Colorado, Boulder, CO 80309-0430.

1 Introduction

The proliferation of parallel and networked computer systems continues to outrace our ability to construct effective software to control such systems. The difficulty can be related to the increased complexity of writing parallel programs compared to that required for writing sequential programs. The added complexity is related to the requirement for managing the activity of simultaneous threads of computation, e.g., see [6].

In this study, we concentrate on two aspects of the problem: the first aspect is related to intuitive mechanisms for perceiving the parallelism in the computation, and the second aspect is in analyzing the resulting programs to determine if they are guaranteed to be functional — or if they are not, then to identify the parts of the program that contribute to the potential nonfunctional behavior.

We have designed a visual parallel programming language, *Phred*, and a support environment that allows a software designer to create Phred programs, to statically analyze them for determinacy, and to interpretively execute them. In this paper, we describe the language (in Section 2) and the analysis mechanism (in Section 3). In companion papers we describe the support system [5] and illustrate the utility of Phred by providing example applications [4].

Parallel software development differs from sequential development in that the programmer must control the creation, destruction, and interaction of parallel tasks. Phred employs the visual aspects of graph models to specify task functionality, and to illustrate points of synchronization and information sharing interactions among tasks. The formal aspects of the graph model provide an unambiguous specification of the interactions that can be analyzed to detect conditions under which the interactions are not guaranteed to be deterministic.

Phred programs are composed of graphs and symbolic interpretations of tasks within the program. Nodes represent units of serial computation and the interconnection of nodes represents control and data flow; the details of the serial units of computation are specified using conventional procedural specifications. Parallelism, synchronization, and data sharing are all specified by the Phred graph, while the computation is specified by a procedure in a conventional language. In this sense a Phred program is both visual and textual.

Since Phred programs represent concurrent activity among tasks, and since tasks may share information repositories, it is possible to construct a Phred program in which two tasks read and write a common information repository. The value left in the repository after the two tasks have completed exe-

cution is determined by the order in which the tasks initiated and completed their execution; determinacy (functionality) can only be guaranteed in the program if tasks that read and write a common information repository will always execute in the same order.

2 The Language

A Phred program is composed of a control flow graph, a data flow graph, and a set of node interpretations. The visual component of Phred is a view of the control and data flow graphs created directly by the programmer, presumably using a mouse-based a graph editor. The node interpretations are represented by procedure specification (using the C programming language in the current implementation).

Arcs in the control flow graph represent disjunctive (OR) and conjunctive (AND) control flow. Thus Phred control flow graph represents sequential flow of control, alternation, parallelism, and synchronization.

Tokens represent the flow of control by marking nodes and edges within the control flow graph. If a token resides on a task node, then this represents the fact that a process is executing the corresponding node interpretation. Tokens also carry data from task node to task node through the control flow graph. When a token arrives at a node, the node procedure is invoked by passing the data contents of the token to the procedure as parameters.

A data repository is a data storage unit which may be accessed by procedures connected to the repository. Therefore data sharing between procedures is explicitly represented by the data flow graph. Data repositories can be viewed as external variables whose scope is explicitly delimited by the arcs in the data flow graph.

Figure 1 is a simple Phred program graph. When a token is placed on the `Start` node, the program is interpreted by assigning a process to the token, and having the process execute the node interpretation for each task on which the token comes to rest. In the example, the start node procedure initializes the `Total` repository to be zero. A token is then sent to the `Get a number` node causing it to execute a procedure that reads an integer, i.e., it prompts the user for an integer. The integer is then placed in a token and sent to the `Odd or Even` node. If the integer in the token is odd it is sent to the `Double` node, otherwise it is sent to the `Half` node, since the interpretation for `Odd or Even` results in a specific control flow path selection based on the data in the token `Double` and `Half` multiply or divide the integer by two respectively. The Output node reads the contents of the `Total` repository, adds the integer from the token to the the

value from the repository, outputs the sum, and writes the new sum back into the `Total` repository. The `Continue` node (procedure) decides whether or not to send the token back for another trip through the loop or to send the token on to the `Stop` node which will terminate the program. `Continue` can make this decision in an arbitrary manner, based on the data available to it (token data in this example).

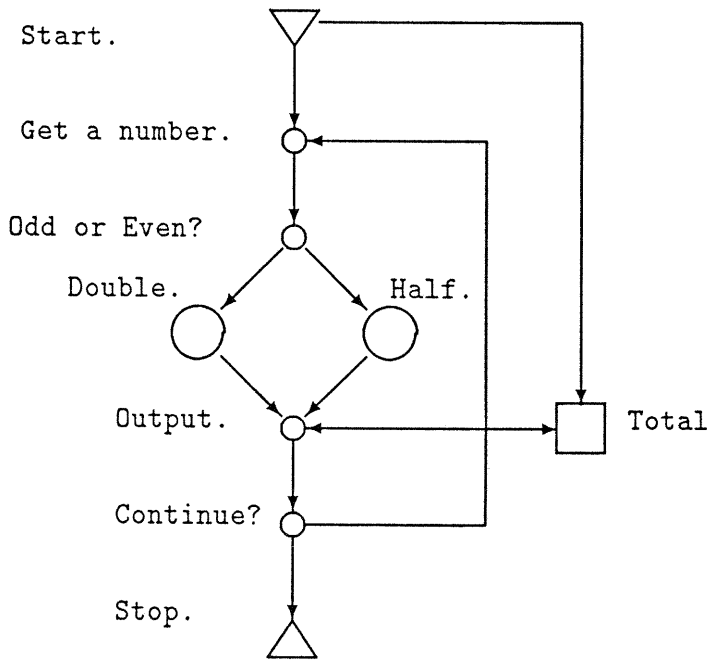


Figure 1: Example Phred program.

This example demonstrates how Phred components (nodes, tokens, and repositories) are used to represent sequential process execution. It contains only disjunctive logic, and has only a single token circulating on the graph at a time. There are other more complex structures to support parallelism, primarily by specifying how multiple tokens may flow through the graph. Based on the intuitive description of Phred semantics, we now turn to a more formal presentation of the components.

2.1 The Building Blocks of Phred

Phred extends the ideas used in the Bilogic Precedence Graph (BPG) model [14], so that they can be applied to visual parallel programming in an MIMD environment. BPGs describe parallel systems and

Phred programs define parallel computations. (PPL is related to CSIM much as Phred is related to BPG [15].)

Phred bears similarities to SCHEDULE [10, 11] in that they both uses graphs to specify the precedence of procedure execution; however, Phred graphs have more semantic functionality built into them than the dependency graphs of SCHEDULE. Phred graphs represent control flow and precedence, while SCHEDULE graphs only represent precedence.

A Phred program explicitly specifies the parallelism of the computation by defining a precedence graph. Nodes in the Phred control flow graph represent functions, and edges between task nodes represent precedence (edges between task nodes and repositories represent data flow). Node interpretations are expressed as functions in any conventional procedural programming language (for simplicity we have restricted our use to C in our implementation of Phred). Default actions are defined for each node type.

Once a Phred program has been specified, it may be executed or interpreted. Mapping the tasks to a parallel machine, passing tokens between tasks, calling user functions when a node is ready to execute, and handling data repositories are performed by an execution environment.

The Phred provides several types of nodes, each with various semantics. (See Figure 2 for the Phred node types.) When the graph is executed, tokens are sent on the control flow input edges. When a token reaches a node, the interpretation is executed. Modifications to the data on an incoming token are unrestricted, with the resulting data from a computation potentially being added to the token and routed to the successor(s). This is similar to Babb and Dinucci's LGDF [1, 2, 9].

Nodes in the control flow graph only have access to the data embedded in tokens and in repositories which are connected to a control flow node. Access to these repositories is specified through the data flow graph. A data repository is a node in the data flow graph that contains shared variables which can be read or written by any node properly connected to the repository in the data flow graph.

Data repositories are read only before the function for a node is executed and written only after the function exits. In addition, two subtypes of repositories, shared and local, are defined. Shared repositories may be accessed from any node in the program. Local repositories have some restrictions, (see [3] for a description of repository access rules). The access patterns for data repositories are essential to determining the determinacy attributes of a Phred program.

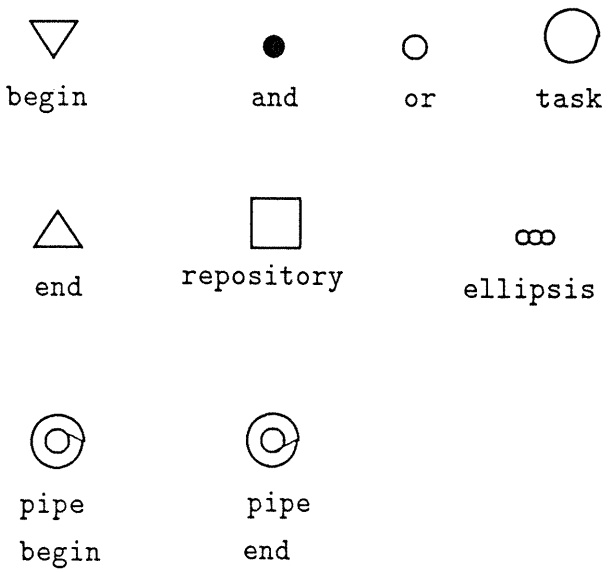


Figure 2: Legal Phred nodes.

2.2 The Grammar

Phred can be formally specified as a graph language using a graph grammar. The following definitions are based on those used by Ehrenfeucht, Main, and Rozenberg in [12]. If Σ is a finite alphabet, then G_Σ denotes the set of all finite directed graphs with nodes from Σ. If α is a graph and v is a node of α of type X, then we call v an X-node. The graph grammar used here is formally defined as follows.

Definition. A graph grammar is a four-tuple (Σ, Δ, P, S), where

- Σ is a finite set of nodes,
- Δ is a proper subset of Σ, called terminal nodes,
- P is a finite set of productions; each production has one of the forms:
  1. $X \rightarrow \alpha$,
  2. $X \rightarrow Seq * \alpha$,
  3. $X \rightarrow \gamma Par * \alpha\varphi$,

where X is a nonterminal node ($X \in \Sigma - \Delta$), γ and φ are terminal nodes ($\gamma, \varphi \in \Delta$), and α is a graph from G_Σ,

- S is a special nonterminal called the start symbol,

Let $G = (\Sigma, \Delta, P, S)$ be a grammar as defined above. Production 1 of P, $X \rightarrow \alpha$, is applied as follows:

1. Start with a graph μ and a specific occurrence of an X-node in μ. This node is called the mother node. The set of nodes directly connected to the mother node is called the neighborhood.
2. Delete the mother node in the graph μ, and call the resulting graph μ'.
3. Add to μ' a copy of the graph α. This new occurrence of α is called the daughter graph.
4. For each node Z in the neighborhood with no outgoing edges, connect a directed edge from Z to every node Y in the daughter graph with no incoming edges.
5. For each node Y in the daughter graph with no outgoing edges, connect a directed edge from Y to every node Z in the neighborhood with no incoming edges.

Production 2 of P, $X \rightarrow Seq * \alpha$, is applied as the production $X \rightarrow \alpha$ except with the following changes to Step 3. Instead of adding one copy of the graph α to μ', n copies of the graph α are added to μ' where $n \geq 0$. The daughter graph consists of graphs $\alpha_1 \ldots \alpha_n$ connected as follows:

- $\forall\, i \in [1, n-1]$, connect each node Z in α_i with no outgoing arcs by a directed edge to each node Y in α_{i+1} with no incoming arcs.

If $n = 0$ then the daughter graph will have no nodes. In this case each node Z in the neighborhood with no outgoing arcs is connected to each node Y in the neighborhood with no incoming arcs.

Production 3 of P, $X \rightarrow \gamma Par * \alpha\varphi$, is also applied as the production $X \rightarrow \alpha$ except with the following changes to Step 3. Instead of adding one copy of the graph α to μ', n copies of the graph α are added to μ', where $n \geq 0$, along with nodes γ and φ. Thus, the daughter graph consists of graphs $\alpha_1 \ldots \alpha_n$ along with nodes γ and φ connected as follows:

- $\forall\, i \in [1, n]$, connect a directed edge from the node γ to each node in α_i which has no incoming edges.

- $\forall\, i \in [1, n]$, connect a directed edge from each node in α_i which has no outgoing edges to the node φ.

If $n = 0$, then the daughter graph has only two nodes, γ and φ. In this case the daughter graph is (γ, φ), the two nodes with a directed edge from γ to φ.

We now use the graph grammar mechanism to describe Phred. Let $\mathbf{PhredCF} = (\Sigma, \Delta, P, S)$ be a graph grammar. The elements of **PhredCF**, Σ, Δ, P, and S are defined as shown in Figure 3.

The **PhredCF** grammar describes the legal control flow graphs in Phred. Several observations should be made about graphs in **PhredCF**. All graphs are single entry, single exit graphs. All statements are also single entry, single exit graphs. The result is that Phred is a highly structured language. All Phred programs must begin with a *Start* node and end with a *Stop* node. Sequences of tasks may be placed wherever a statement is legal. *Repeat-until* loops and *While* loops may be constructed with the loop construct.

The *Pipe* construct allows statements within it to be pipelined. Multiple tokens are emitted from the *Begin Pipe* node. These tokens may cause several tasks within the *Pipe* construct to execute in parallel, even though there is an edge in the control flow graph between them. Therefore within a *Pipe* construct, the control flow graph no longer represents precedence, but simply the pipelined flow of tokens through the statements within the construct.

There are two types of conjunctive constructs. The *Manual Conjunctive* construct allows the programmer to place a fixed number of heterogeneous statements within the construct. This is useful for specifying a small number of statements that may execute in parallel, i.e. functional parallelism. The *Dynamic Conjunctive* construct allows the programmer to specify a number of homogeneous statements to execute in parallel. The number of statements to execute is determined at run-time by calling a function from the diverging conjunctive node. This is part of the interface between the node routines and the graph which is discussed in detail in described in [3].

The disjunctive constructs are similar to the conjunctive constructs. The *Manual Disjunctive* construct allows the programmer to specify several branches which may be taken. This could be used for a conditional `if` or `case` statement. The *Dynamic Disjunctive* structure will allow the programmer to dynamically specify the number of branches to create at run-time. The statements in the *Dynamic Disjunctive* are homogeneous. If this construct were put within a pipe construct, the diverging *Or* node, at the beginning of the construct, could decide to create a branch for each token in the pipe structure, providing data parallelism within the pipe structure. The *null* statement is also a legal Phred statement.

$\Sigma = \{Prog, Conj, Disj, Stmt, Pipe, Loop\} \cup \Delta,$
$S = Prog$

$\Delta = \{$  $\}$

$P =$

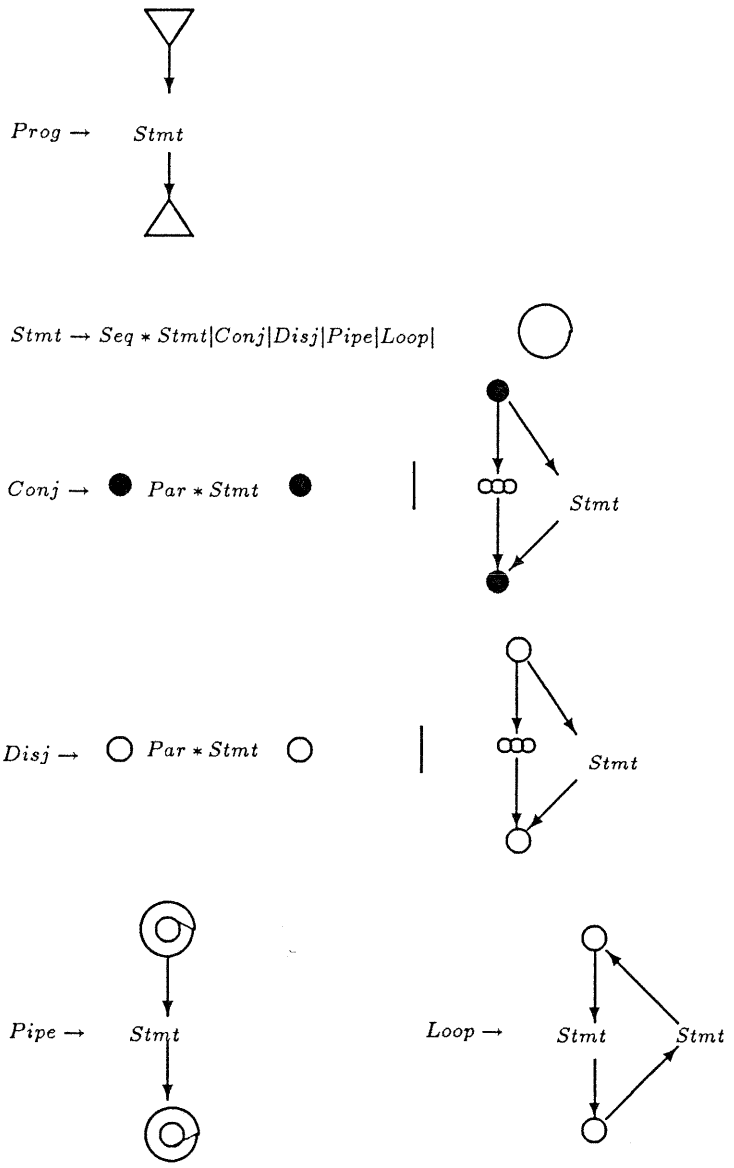


Figure 3: Phred grammar.

A Phred graph may contain repository nodes which are connected to the control flow graph. **PhredCF** describes the structure of the Phred control flow graphs. The Phred dataflow graphs have less structure than the control flow graphs. Any number of repository nodes may be added to a Phred graph. Directed edges may connect a repository node to any nodes in the control flow graph. Edges may not connect repository nodes to each other. An edge from a repository to a control flow node indicates data in the repository may be read by the control flow node. An edge to a repository from a control flow node indicates data may be written to the repository by the control flow node.

2.2.1 An Example Derivation

Figure 4 is a derivation of a graph in the Phred language. Various productions are applied to the nonterminal

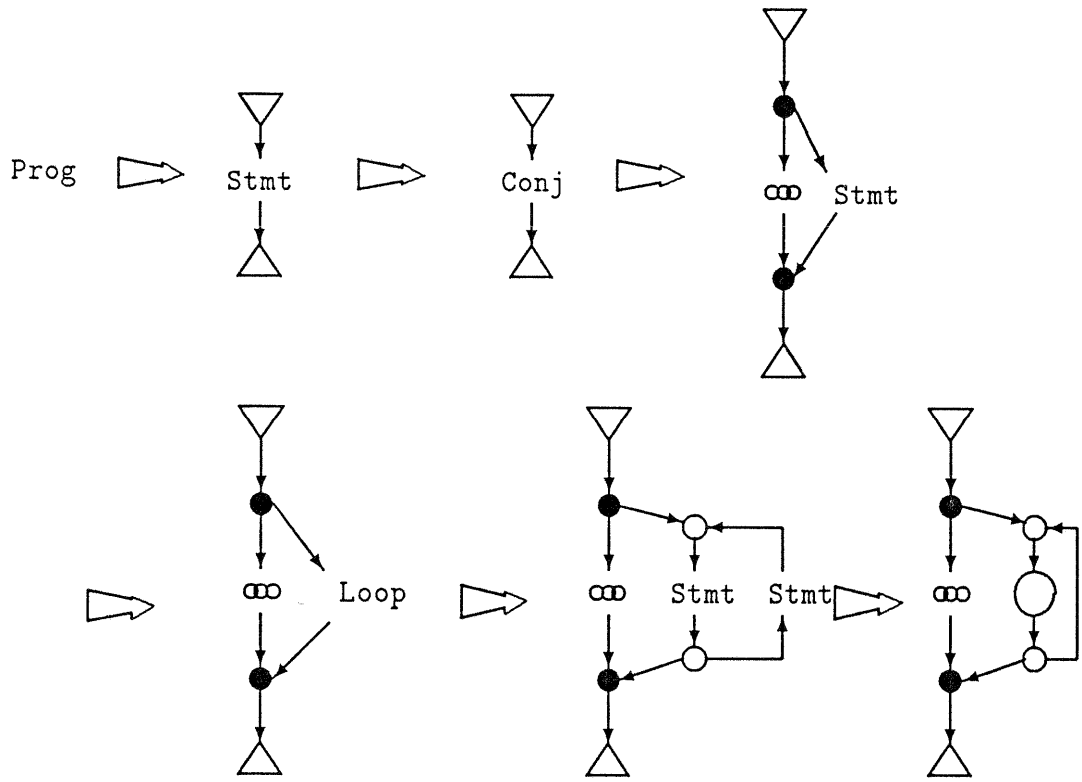


Figure 4: Example graph derivation.

start symbol `Prog`. The nonterminal symbol `Prog` is first replaced by the `Stmt` nonterminal bounded by a

Start node and an **End** node. The `Stmt` is then replaced by a `Conj` nonterminal and so on, resulting in the final graph.

2.3 A Program Example

Another example helps to tie together these concepts: Figure 5 shows a Phred graph complete with data repositories. This program computes the values for the vector x as a solution to $Ax = b$, where A and b are

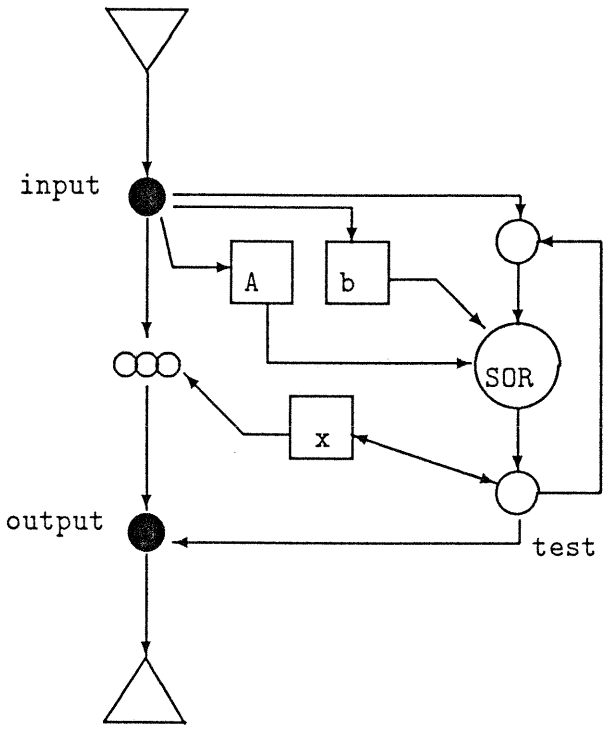


Figure 5: Example Phred program graph.

matrices. When this Phred program is executed, a token is placed on the *Start* node. After the *Start* node executes, the *And* node labeled `input` executes. This node writes the matrices A and b to repositories A and b. Since these repositories are accessed by the `input` node, which is not dynamically created, A and b are shared repositories. The code for `input` also specifies that n copies of the loop containing the `SOR` node be created, one to compute each element of x. Note that n x repositories will be be created along with each copy of the loop. This is done by making a call to the `create_tokens()` routine. This is how Phred

program graphs may dynamically change at run-time. A token is then dispatched to each copy of the loop. This token traverses the body of the loop, executing the `SOR` node each time. The `SOR` node reads the `A`, `b` and `x` repositories and computes a new value for its element of the x vector. If the `test` node determines that another iteration is necessary, the loop will continue. Otherwise, the loop terminates and the token is sent to the `output` node. When tokens from all the test nodes arrive at the `output` node, it executes, outputting the results of the computation, i.e. the elements of the x vector. The token is then be sent to the *Stop* node and the program terminates.

3 Analysis of the Language

The Phred grammar and semantic model allow one to analyze a program for correct syntax and for determinacy. In this Section, we describe the analysis.

3.1 Remarks about Determinacy

When we learn to program, the idea of determinacy begins to develop. We expect our programs, no matter how unusual, to be faithfully executed by the computer. If we run a program over and over on the same input set, we expect to repeatedly get identical output. This becomes a fundamental property for debugging programs and using computers in general. Based on this experience, an informal definition of determinacy arises. Loosely defined, a system is determinate if each time it is run on an input it produces the same output. This is the same as the mathematical definition of a function: a unique output set is defined for each input set.

What are some of the advantages and disadvantages of determinacy? Requiring a system to be determinate may result in an efficiency loss. This follows from the possibility that determinacy may require unnecessary synchronization in the program, resulting in costly blocking delays. Although nondeterminacy may seem advantageous in terms of efficiency, it can make the programming task more difficult. What are the consequences of nondeterministic programs or systems in general? Bernstein and Schneider [7] state "programs that do not exhibit reproducible behavior are very difficult to understand and validate." It is, indeed, easy to see the problems in debugging a nondeterminate ("time dependent") program or system. It may be very difficult to isolate a bug that is not easily reproducible.

Testing is a related issue. How does one test a system that is nondeterminate? If the program is allowed to output different answers each time it is executed then not only will it need to be tested on many different inputs but it will in fact need to be tested many times on the same input. Even then, there is no assurance that the program is correct for a specific input set. In general, nondeterminacy may be conceptually useful or a practical necessity, but there are certain costs associated with nondeterminacy which should be considered.

Determinacy can be formally defined in terms of mutually noninterfering tasks. A pair of tasks are mutually noninterfering if they either 1) do not execute in parallel or 2) do not violate Bernstein's conditions. Bernstein's conditions are the necessary and sufficient conditions needed to ensure determinate operation of a pair of parallel tasks. They state that tasks which can execute in parallel could be nondeterministic if they use or change a shared resource. Normally the shared resource is memory or a file system. Theorem 1 shows the necessary and sufficient conditions for a set of tasks to execute deterministically. The line marked with the dagger (†) identifies Bernstein's conditions. A formal proof of the following theorem can be found in [8].

Theorem 1 *Conditions for mutually noninterfering tasks.*

- *Let t and t' be tasks which may run in parallel.*
- *Let W_t be the write set of a task t.*
- *Let R_t be the read set of a task t.*
- *Tasks t and t' are said to be noninterfering $\iff$ either:*
  - *t is the successor or predecessor of t' or*
  - $(W_t \cap W_{t'} = \emptyset) \wedge (R_t \cap W_{t'} = \emptyset) \wedge (W_t \cap R_{t'} = \emptyset)^{\dagger}$

A set of tasks $\{t_1, ..., t_n\}$ is determinate $\iff$ t_i and t_j are mutually noninterfering $\forall\, i, j \in [1, n]$ where $i \neq j$.

Nondeterminacy may result from a violation of Bernstein's conditions. An example of this would be when two tasks write different values to a shared memory location in parallel. This may be the result of some precedence relation being accidently omitted or an intentional nondeterministic action deliberately placed

in the program. While the interference of parallel or concurrent tasks is the crux of nondeterminacy, such actions may not necessarily cause nondeterminacy. For instance, if two parallel tasks write the same value to a shared location in parallel it is a violation of Bernstein's conditions yet the tasks are still determinate. Therefore the violation of Bernstein's conditions indicates the *possibility* of nondeterminacy but does not the guarantee that nondeterminacy is present. That is, all nondeterministic tasks violate Bernstein's conditions, but not all violations of Bernstein's conditions result in nondeterminacy.

3.2 Determinacy in Phred Graphs

The issue of determinacy must be addressed by parallel programming languages. The reason determinacy is important in parallel programming is well-stated by LeBlanc and Mellor-Crummey [13]:

> Since parallel programs do not fully specify all possible execution sequences, the execution behavior of a parallel program in response to a fixed input may be indeterminate, with the results depending on a particular resolution of race conditions existing among processes.

If a computation is specified for a parallel machine, there are chances for interference between the parallel tasks specified by the program. Much effort in parallel languages is spent on providing useful mechanisms for ensuring determinate behavior of parallel programs. In fact the different mechanisms for ensuring determinacy provided by parallel languages, to a great extent, distinguish such languages from each other.

It is interesting to note that although many parallel languages provide mechanisms to ensure determinacy, they do not, in general, enforce determinacy as a requirement.

The main emphasis of Phred is to develop a parallel programming environment which encourages deterministic programs without requiring programs to be deterministic. Given the definition of Phred, we will now explore the conditions under which Phred programs are deterministic. If a program is found to be nondeterministic, then repositories and tasks involved in the nondeterminacy can be identified.

3.3 Identifying Nondeterminacy in Phred Programs

Determinacy is defined in terms of task access to shared repositories. In Phred, nondeterminacy may occur when tasks that may execute in parallel share repositories in ways which violate Bernstein's conditions (Theorem 1). More precisely, if the set of tasks which make up a Phred program are mutually noninterfering

according to Theorem 1, the program is determinate. If these tasks are not mutually noninterfering then the program may be nondeterminate. Tasks which violate Bernstein's conditions may still be determinate under certain conditions. Algorithm 1 finds tasks and repositories which may make a Phred program nondeterminate.

Algorithm 1 *Checking for determinacy.*

- *Let P be the set of sets of nodes which could possibly execute in parallel.*
- *Let Rep be the set of all repositories.*
- *Let $r \in Rep$*
- *Let R_r be the set of tasks which read repository r.*
- *Let W_r be the set of tasks which write repository r.*

Let $T_n \Leftarrow \emptyset$ and $R_n \Leftarrow \emptyset$.
For each $r \in Rep$ and each $p \in P$:

$$\text{Let } X \Leftarrow (W_r \cap p) \cup (R_r \cap p)$$

$$T_n \Leftarrow \begin{cases} T_n \cup (W_r \cap p) & \text{if } \| W_r \cap p \| > 1 \\ T_n \cup X & \text{if } (W_r \cap p \neq \emptyset) \wedge (R_r \cap p \neq \emptyset) \wedge (\| X \| > 1) \end{cases}$$

$$R_n \Leftarrow \begin{cases} R_n \cup \{r\} & \text{if } \| W_r \cap p \| > 1 \\ R_n \cup \{r\} & \text{if } (W_r \cap p \neq \emptyset) \wedge (R_r \cap p \neq \emptyset) \wedge (\| X \| > 1) \end{cases}$$

The goal is to find the tasks and repositories, if any, that might cause nondeterminacy in a program. Algorithm 1 builds two sets. T_n is the set of task nodes in the program which are involved in a violation of Bernstein's conditions and thus may contribute to any nondeterminacy. R_n is the set of repositories in the program which may contribute to any nondeterminacy. The algorithm chooses a particular repository and a set of tasks which may execute in parallel. If any subset of these tasks violate Bernstein's conditions in their access of the repository then they are possibly nondeterminate. If tasks access a repository in this manner then they are added to T_n and the repository is added to R_n. The elements of P are themselves sets. Each set, $p \in P$, contains tasks which may execute in parallel. The algorithm checks each repository,

$r \in Rep$, against the elements of P. The set X contains task nodes of p which read or write a given repository r. If there is more than one task from p that writes r then those tasks are added to T_n. If there is at least one task in p which writes to r and at least one task in p that reads from r and these tasks are not the same ($\| X \| > 1$) then these tasks are added to T_n. Repositories are added to R_n under the same conditions. Thus, the set T_n contains the task nodes which could contribute to any nondeterminacy and R_n the repositories involved in this possible nondeterminacy.

There are three assumptions made with respect to determinacy checking:

1. The set of sets of tasks which could possibly execute in parallel, P, can be found.

2. Tasks do not share any resources other than repositories.

3. The task nodes are deterministic.

Assumption 2 forbids the user code associated with task nodes from sharing data through means other than those specified by the graph. Assumption 3 excludes the possibility of the programmer including nondeterministic code within a task node, i.e., the use of uninitialized variables.

Theorem 2 *A Phred program is determinate* $\iff T_n = \emptyset$.

Theorem 1 states that a set of tasks is determinate (mutually noninterfering) if and only if the tasks which may execute in parallel neither write a shared resource in parallel nor do they read and write a shared resource in parallel. Since T_n is the set of tasks which either write a shared resource in parallel or read and write a shared resource in parallel, one can conclude from Theorem 1 that a Phred program is determinate if and only if $T_n = \emptyset$ after all $(r, p) \in Rep \times P$ have been analyzed.

Theorem 3 $T_n = \emptyset \iff R_n = \emptyset$.

Both T_n and R_n are initially the empty set and elements are added to both sets under the same conditions, namely when parallel tasks access a repository in a manner which violates Bernstein's conditions. Therefore $T_n = \emptyset \Rightarrow R_n = \emptyset$ and $R_n = \emptyset \Rightarrow T_n = \emptyset$, so, $T_n = \emptyset \iff R_n = \emptyset$.

Theorem 4 *If a Phred program is not determinate then* T_n *is the set of tasks which cause the nondeterminacy.*

Tasks are in the set T_n precisely because they violate Bernstein's conditions. Thus it can be seen that Theorem 4 is also true.

Theorem 5 *If a Phred program has no repositories, then it is determinate.*

If a program has no repositories ($Rep = \emptyset$), then we can also infer that $R_n = \emptyset$ since $R_n \subseteq Rep$. If $R_n = \emptyset$ then by Theorem 3, $T_n = \emptyset$. Hence, by Theorem 2, the program with no repositories is determinate since $T_n = \emptyset$. Therefore Theorem 5 is true. Intuitively one can see that Theorem 5 is true; if a program has no repositories, then there is no shared resource to be accessed nondeterministically by parallel tasks.

It has now been shown that Phred programs may be tested for nondeterminacy. Furthermore, the tasks and repositories involved in the nondeterminacy will be identified by the process. Next it must be shown that P, the set whose elements are sets of tasks which may execute in parallel, can be found (Assumption 1).

3.3.1 Finding Parallel Tasks

The goal is to find all sets of nodes which could possibly run in parallel, that is, the set P from Algorithm 1. In general, tasks which are on different branches of the same disjunctive node could possibly run in parallel. Nodes within a pipe construct may also run in parallel, but this case is dealt with later. Tracing the path to a node through its conjunctive ancestors is the determinative feature of the algorithms presented here. Algorithm 2 is used to mark the nodes of a graph with labels which will be used by Algorithm 3 to test if two nodes may execute in parallel.

Algorithm 2 *Marking Phred graphs.*

```
mark(n,s)

if n is already marked /* n_mark ≠ ∅ */
    return

if n is a diverging conjunctive node
    /* # inarcs = 1 and # outarcs > 1 */
    n_mark ← s ← concat(s,id)
    for i ← 1 to number_outarcs
            mark(succ_i, concat(s,i))
    return
```

```
if n is a converging conjunctive node
    /* # inarcs > 1 and # outarcs = 1 */
    n_mark = head(s, length(s)-1)
    mark(succ_n, head(n_mark, length(n_mark)-1))
    return

if n is a terminator /* #outarcs = 0*/
    n_mark = s
    return

/* otherwise mark this node and its successors with s */
n_mark = s
for i = 1 to number_outarcs
    mark(succ_i, s)
return
```

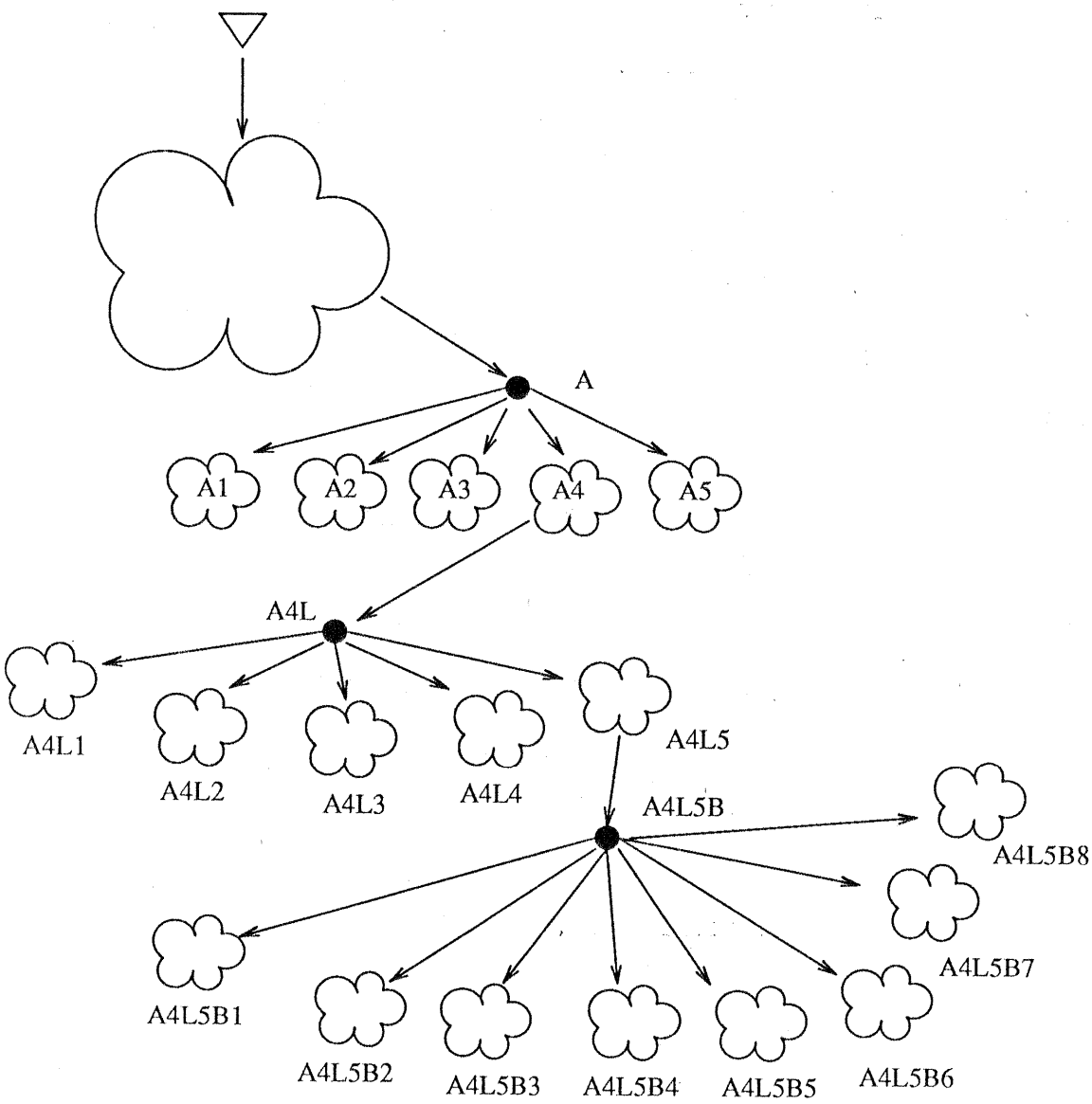
Algorithm 2 recursively traverses the graph, marking each node in the graph with a string. This string is generated by the conjunctive branches taken to arrive at a node. Thus, this string is in one sense a history of the branches a token must have taken to reach a particular node. In Algorithm 2, `succ(i)` is the node at the end of the i^{th} out arc. Also, `id` is the unique id of the node being marked.

The key to understanding the marking algorithm is that diverging conjunctive nodes are the only nodes that may specify parallelism (excluding pipe nodes which are dealt with differently since pipe nodes remove precedence temporarily within their scope). The algorithm marks a node with a string indicating the branches from its ancestral conjunctive nodes which were taken to reach the node. For instance, the marking **A4L5B8** indicates several things. A node with such a marking has conjunctive nodes **A**, **L** and **B** as ancestors. Furthermore node **A** is an ancestor of node **L** and node **L** is an ancestor of node **B**. A node with such a marking can be found on the fourth branch from node **A** followed by the fifth branch from node **L** and the eighth branch from node **B**. Figure 6 illustrates this marking scheme. Diverging conjunctive

(see next page)

Figure 6: Conceptual view of markings based on conjunctive node branches.

nodes themselves are marked with their predecessor's marking plus their unique node label, but no branch indicator (since a conjunctive node is not on any of its own branches). A converging conjunctive node is labeled in the same manner as its matching diverging conjunctive node. All other nodes are simply marked with the same string as their immediate predecessor. A complete Phred graph with markings can be seen


A
A1
A2
A3
A4
A5
A4L
A4L1
A4L2
A4L3
A4L4
A4L5
A4L5B
A4L5B8
A4L5B7
A4L5B1
A4L5B2
A4L5B3
A4L5B4
A4L5B5
A4L5B6

in Figure 7.

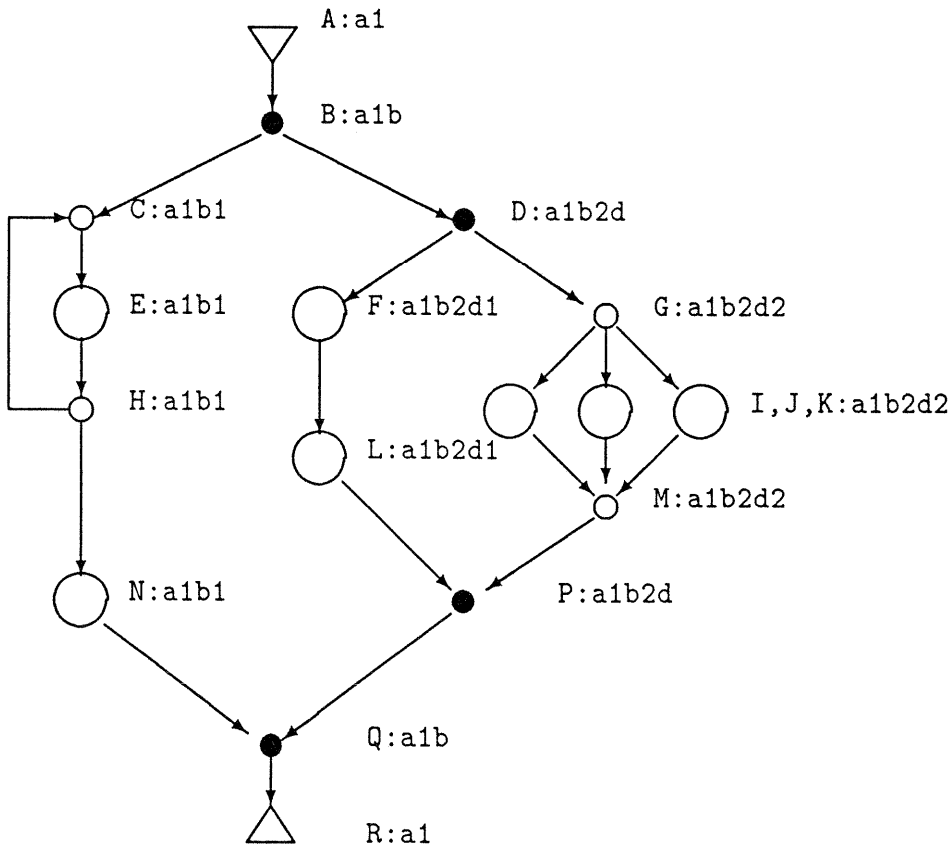


Figure 7: Example Phred graph with markings based on conjunctive node branches.

Once all nodes in a Phred graph are marked in the manner described here, the question of whether two nodes may run in parallel can be answered. Algorithm 3 checks two nodes to see if they might be executed in parallel. The marking strings for the nodes are compared from left to right, one character at a time. If the node identification string is different, then the two nodes cannot run in parallel since the nodes do not share a common conjunctive ancestor, and the checking may stop. If the node id is the same then the checking must continue with the branch number. If the node id is the same, but the branch number is different, then the nodes could execute in parallel. If the node id is the same and the branch is the same then checking must continue with the next node id and branch number. If there are no more markings

left to check and the question of parallel execution has not been decided, then the nodes cannot execute in parallel.

Algorithm 3 *Testing for Parallel Execution*

```
/* a and b are the marking strings for two nodes */

/* check each character in the strings a and b */
i ← 0
while i < min(length(a),length(b))
        /* if the node is different */
        if a_i ≠ b_i
                return SERIAL /* then return SERIAL */
        i ← i + 1
        /* if there is no more string left to check */
        if i ≥ min(length(a),length(b))
                return SERIAL /* it must be SERIAL */
        /* if the node is the same but
        the branch is different */
        if a_i ≠ b_i
                return PARALLEL /* it must be PARALLEL */
        i ← i + 1
/* it must be serial */
return (SERIAL);
```

Algorithms 2 and 3 will find nodes which may run in parallel because they are on different branches of a common conjunctive node. This is not enough to tell if two nodes may run in parallel. Nodes must be checked to see if they are within the same pipe construct as well. This can also be done by marking the graph and then comparing marking strings. Such a marking scheme can be carried out since pipe constructs are always properly nested. Nodes in the graph are marked with a string showing which pipe constructs contain the node. For instance, a node X with a marking of **ALB** would indicate that the node is nested within pipe constructs **A**, **L**, and **B**. In such a case X could possibly execute in parallel with any node whose pipe marking was a prefix of **ALB**. Similarly any node which has **ALB** as a prefix of its marking could also execute in parallel with X. Figure 8 shows a Phred graph containing nested pipe structures and the pipe marking strings. Since the label for node C, **b**, is a prefix of the label for node E, **bd**, it is possible for nodes C and E to run in parallel. Special rules apply to begin- and end-pipe nodes. If the marking of a

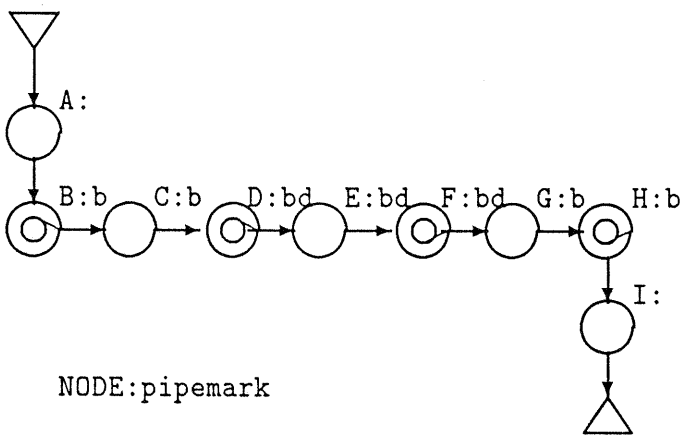


Figure 8: Example labels for pipe marking.

begin- or end-pipe node have the exact same label as another node, then the nodes cannot run in parallel. For example in Figure 8, nodes B and C cannot run in parallel since B is a begin-pipe and C is a node within its structure. If the marking for a begin- or end-pipe node is a prefix of the marking for another node, then these two nodes cannot execute in parallel either. In Figure 8, node H cannot run in parallel with node E since H is an end pipe node and E is a node within its structure. These rules are consistent with the definition of the *Pipe* construct.

Using the pipe marking scheme in conjunction with Algorithms 2 and 3, the set P, a set of sets of tasks which may run in parallel, can be found for a Phred graph. Note that the conjunctive marking and the pipe marking can be carried out in a single recursive pass over the graph. Once P has been found, Algorithm 1 can be used to find any nondeterministic tasks and repositories in the Phred program.

Three assumptions were stated at the beginning of Section 3.3. The first assumption is that the set P can be found. Methods for building P are given previously. The second assumption is that Phred nodes do not share any resources other than those shown in the Phred graph as a repository access. Since control flow nodes in the graph represent procedures in a conventional language, it is possible that these procedures call system routines which would violate assumption two. For instance, if two node interpretations share variables through some low level system calls, all analysis at the graph level is meaningless. The programmer must use only Phred constructs for specifying parallelism and sharing data. The third assumption, that the task nodes are determinate, is equivalent to good programming habits in serial code. This assumption

is no more grave than stating that irresponsible use of things like uninitialized variables may invalidate the determinacy checking at a higher level. Therefore, we have now shown that given several reasonable assumptions Phred programs can be shown to be determinate.

4 Conclusion

The importance of determinacy in parallel programs can easily be overlooked by programmers moving between the worlds of sequential and parallel programming. In certain cases, algorithm designers rely on nondeterminacy to avoid unnecessary decision making (and synchronization), since it is known that an overall correct result can be obtained without retaining functional behavior. Selecting a particular strategy in a game-playing algorithm is a good example of this approach.

However, we believe that determinacy is generally desirable, particularly in the large collection of parallel *application* programs that must begin to be implemented (to justify the extensive investment in hardware). We also speculate that race conditions in nondeterministic situations will become a major intellectual stumbling block for parallel programmers, much as pointer are a major stumbling block for C programmers. Just as strongly-typed languages attack the pointer problem, Phred attacks the nondeterminacy problem.

The graph model is the basis of the determinacy analysis. It also provides a natural user interface for visual programming. Here, we believe that the top-down approach of designing control and data flow (prior to supplying interpretations) will be naturally supported by the Phred graph. We expect that the software designer will use the graphs to qualitatively explore algorithms, then annotate the graphs to produce deterministic programs.

Phred has been carefully designed to incorporate components that have been shown to be useful for encoding parallel algorithms in other languages, and which can be combined in a manner that allows us to check the program for determinacy. We do not make any strong argument for the completeness of the language, except to point out that it incorporates primitives found in many other parallel languages. The companion paper and [3] illustrate Phred's utility by showing a selection of parallel programs expressed in Phred.

This paper describes the Phred programming language; it is only part of our study on the subject. Another part of the work has been to implement a support system for Phred. The implementation work, described in [5], has resulted in a system that can be used to interactively construct Phred programs, to

parse them and analyze them as they are being created (using parallel implementation techniques to avoid the annoyances associated with a syntax-directed editor), and to interpret them using the BPG execution environment.

References

[1] R. B. Babb II and D. C. DiNucci. Design and Implementation of Parallel Programs with Large-Grain Data Flow. In D. B. Gannon L. H. Jamieson and R. J. Douglass, editors, *The Characteristics of Parallel Algorithms*, chapter 13. The MIT Press, Cambridge, Massachusetts, 1987.

[2] R. G. Babb II. Parallel processing with large-grain data flow techniques. *Computer*, pages 55–61, July 1984.

[3] A. Beguelin. *Deterministic Parallel Programming in Phred.* PhD thesis, University of Colorado, Department of Computer Science, Boulder, Colorado 80309-0430, 1990.

[4] A. Beguelin and G. Nutt. Examples in Phred, 1990. Work in progress.

[5] A. Beguelin and G. Nutt. A tool and language for visual distributed programming, 1990. Work in progress.

[6] G. Bell. The future of high performance computers in science and engineering. *Communications of the ACM*, 32(9):1091–1101, September 1989.

[7] A. J. Bernstein and F. B. Schneider. On Restrictions to Ensure Reproducible Behavior in Concurrent Programs. Technical report, Department of Computer Science, Cornell University, 1979.

[8] E. G. Coffman and P.J. Denning. *Operating System Theory.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1973.

[9] D. C. DiNucci and R. G. Babb II. Design and implementation of parallel programs with LGDF2. In *IEEE COMPCON Spring*, pages 102–107, 1989.

[10] J. J. Dongarra and D. C. Sorensen. A portable environment for developing parallel FORTRAN programs. In *Proceedings of the International Conference on Vector and Parallel Computing - Issues in Applied Research and Development*, pages 175–186, July 1987. Published in Parallel Computing, Volume 5, Numbers 1 & 2.

[11] J. J. Dongarra and D. C. Sorensen. SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs. In D. B. Gannon L. H. Jamieson and R. J. Douglass, editors, *The Characteristics of Parallel Algorithms*, pages 363–394. The MIT Press, Cambridge, Massachusetts, 1987.

[12] A. Ehrenfeucht, M. G. Main, and G. Rozenberg. Restrictions on NLC graph grammars. *Theoretical Computer Science*, 31:211–223, 1984.

[13] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, April 1987.

[14] G. Nutt, A. Beguelin, I. Demeure, S. Elliott, J. McWhirter, and B. Sanders. Olympus: An interactive simulation system. In Edward A. MacNair, Kenneth J. Musselman, and Philip Heidelberger, editors, *1989 Winter Simulation Conference Proceedings*, pages 601–611, December 1989.

[15] H. Schwetman. *PPL Reference Manual.* Microelectronics and Computer Technology Corporation.

FOOTNOTES

* This work was supported by NSF Grant CCR-8802283.

† This work was done while this author was at the University of Colorado, Boulder.

‡ Please direct correspondence to: Professor Gary Nutt, Department of Computer Science, University of Colorado, Boulder, CO 80309-0430.

INDEX TERMS

Index Terms: determinacy, graph grammars, graph models, graph parsing, parallel computing, parallel programming, programming environments, programming langauges, static analysis, visual programming.

PREFERRED ADDRESS

Professor Gary Nutt

Department of Computer Science

Campus Box 430

University of Colorado

Boulder, CO 80309-0430

FIGURE CAPTIONS

Examples in Phred *

Adam Beguelin †
Department of Computer Science
University of Tennessee
Knoxville, TN 37996–1301
adamb@cs.utk.edu

Gary Nutt ‡
Department of Computer Science
Campus Box #430
University of Colorado
Boulder, CO 80309–0430
garyn@boulder.colorado.edu

Abstract

Phred is an experimental visual parallel programming language, intended to assist programmers in designing and implementing parallel programs by providing a natural, visual interface and by providing facilities to identify potentially nondeterministic parts of a computation. This paper demonstrates the utility of the language by providing examples of parallel scientific applications written in Phred. Variations of chaotic and successive over relaxation methods are presented to compare and contrast their synchronization methods; the differences are visually apparent in the Phred implementations. We also present a Phred version of parallel adaptive global optimization, to illustrate the language's use in adaptive algorithms.

1 Introduction

The purpose of this paper is to present several complex parallel numerical algorithms in the Phred programming language. Phred is a hybrid programming language, combining both textual and visual programming. Phred is a structured programming language. Control flow is restricted in a Pascal-like fashion. The visual component of Phred is a structured graph, defined formally by a graph grammar, which describes the control flow, parallelism, synchronization, and data flow of a computation. Nodes in the Phred control flow graph represent procedures specified textually

*This report is a preliminary version of a paper presented at the Fifth SIAM Conference on Parallel Processing for Scientific Computing, Houston, Texas, March, 1991.

†This work was done while this author was at the University of Colorado, Boulder (supported by NSF Grant No. CCR-8802283), and the Ecole Nationale Supérieure des Télécommunications, Paris.

‡Supported by NSF Grant No. CCR-8802283.

in a conventional programming language such as Fortran or C. Certain nodes in the Phred data flow graph, called data repositories, represent data shared among nodes in the control flow graph. The explicit representation of shared data in the Phred program makes it possible for determinacy analysis to be performed on Phred programs. Thus it is possible to automatically detect points in the program where nondeterminacy may occur. This nondeterminacy information may be useful for both the analysis and debugging of a parallel computation.

After briefly describing the Phred programming language, three in depth parallel numerical examples are presented. The first two programs iterativly solve a system of linear equations. One by a parallel synchronous successive over relaxation method and the other via a parallel chaotic method. Given the linear system $Ax = b$ where A and b are known, these algorithms solve for x. Successive x vectors are computed, $x_0, x_1, x_2, \ldots$, such that x_n approaches x as n grows. These two iterative algorithms differ mainly in the placement of synchronization points. This difference is visually apparent in the two Phred implementations. The synchronous SOR method updates the elements of the x vector synchronously upon each iteration. In the chaotic method iterations are not synchronized, elements of the x vector may be updated by processes executing various iterations.

While the first two examples are standard numerical fare, the final example is taken from current research on adaptive global optimization [12]. The algorithm is a parallel iterative stochastic method for finding the minimizers of a function in some space S. In the parallel version, S is subdivided into regions which are operated upon in parallel. These regions may be further subdivided, resulting in the adaptive behavior of the algorithm. This example further demonstrates how Phred can be used to visually describe complex parallel algorithms.

2 A Brief Overview of Phred

Although visual programming is not the major focus of this research, Phred is a visual programming language. The visual aspects of Phred allow the programmer to graphically see the parallelism, control flow, and data flow facets of a program. While this paper does not attempt to address the merits of such visualization, others argue several advantages to visual programming [10, 11]. The programmer, with the aid of the Phred programming environment described in [3, 1], can immediately see which parts of a program may cause nondeterminacy. This graphical specification also lends itself to animation of program execution which would be useful in debugging or optimizing Phred programs [6, 8, 9].

The **PhredCF** grammar, Figure 1, describes the legal control flow graphs in Phred [3, 2].

Several observations should be made about graphs in **PhredCF**. All graphs are single entry, single exit graphs. All statements are also single entry, single exit graphs. The result is that Phred is a highly structured language. All Phred programs must begin with a *Start* node and end with a *Stop* node. Sequences of tasks may be placed wherever a statement is legal. *Repeat-until* loops and *While* loops may be constructed with the loop construct.

The *Pipe* construct allows statements within it to be pipelined. Multiple tokens are emitted from the *Begin Pipe* node. These tokens may cause several tasks within the *Pipe* construct to execute in parallel, even though there is an edge in the control flow graph between them. Therefore within a *Pipe* construct, the control flow graph no longer represents precedence, but simply the pipelined

$\Sigma = \{Prog, Conj, Disj, Stmt, Pipe, Loop\} \cup \Delta,$
$S = Prog$

$\Delta = \{$  $\}$

$P =$

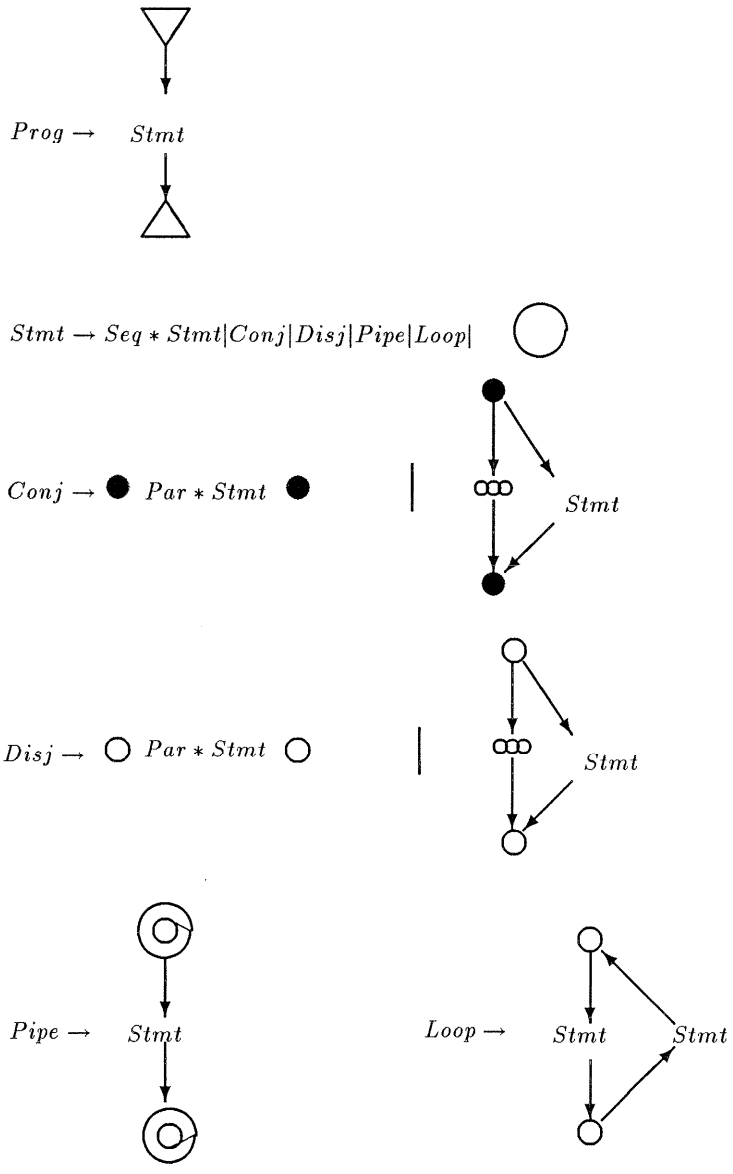


Figure 1: Phred grammar.

flow of tokens through the statements within the construct.

There are two types of conjunctive constructs. The *Manual Conjunctive* construct allows the programmer to place a fixed number of heterogeneous statements within the construct. This is useful for specifying a small number of statements that may execute in parallel, i.e. functional parallelism. The *Dynamic Conjunctive* construct allows the programmer to specify a number of homogeneous statements to execute in parallel. The number of statements to execute is determined at run-time by calling a function from the diverging conjunctive node, (see Appendix A for more details).

The disjunctive constructs are similar to the conjunctive constructs. The *Manual Disjunctive* construct allows the programmer to specify several branches which may be taken. This could be used for a conditional `if` or `case` statement. The *Dynamic Disjunctive* structure will allow the programmer to dynamically specify the number of branches to create at run-time. The statements in the *Dynamic Disjunctive* are homogeneous. If this construct were put within a pipe construct, the diverging *Or* node, at the beginning of the construct, could decide to create a branch for each token in the pipe structure, providing data parallelism within the pipe structure. The *null* statement is also a legal Phred statement.

A Phred graph may contain repository nodes which are connected to the control flow graph. **PhredCF** describes the structure of the Phred control flow graphs. The Phred dataflow graphs have less structure than the control flow graphs. Any number of repository nodes may be added to a Phred graph. Directed edges may connect a repository node to any nodes in the control flow graph. Edges may not connect repository nodes to each other. An edge from a repository to a control flow node indicates data in the repository may be read by the control flow node. An edge to a repository from a control flow node indicates data may be written to the repository by the control flow node.

2.1 Types of Parallelism

Phred allows the programmer to specify several important types of parallelism [7]. Data parallelism, functional parallelism and pipelining are all easily expressible in Phred. Data parallelism can be most simply specified by using the dynamic conjunctive construct. It allows the programmer to specify a number of data items to be computed by the same function in parallel. Data parallelism may also be specified by using a disjunctive construct within a pipe construct. Functional parallelism can be specified by using the manual conjunctive construct or the manual disjunctive construct within a pipe construct. Functions may be pipelined by placing them sequentially within a pipe construct. Producer-consumer parallelism can be specified with the pipe construct as well. These are the main types of parallelism that Phred programmers can use.

3 Parallel Successive Over-Relaxation

The parallel iterative successive over-relaxation algorithm (SOR) is used solving $Ax = b$ for x. A is an $n \times n$ matrix. Vectors x and b are of length n. This chaotic parallel version of the successive over-relaxation algorithm (SOR) is based on the algorithm in [4]. Elements of x are computed iteratively from A, b, and the elements of x from a previous iteration. Iteration continues until

either the values of x converge to within some tolerance or some maximum number of iterations is reached. The Phred graph for this algorithm can be found in Figure 5 and the C code is in Figures 2, 3, and 4.[1]

The start node reads the matrix A and the vector b, (see Figure 2). These are placed into a shared repository to be read by each SOR task node. The start node also initializes n subgraphs, one to compute each element of the vector x. The token sent to the SOR task node is initialized with the user-supplied maximum number of iterations for the computation.

The local repository X contains the vector x to be computed. Since it is a local repository, one copy is created for each SOR task. Each SOR node has read and write access to its X repository and read access to all other X repositories. The SOR task computes a new x element and stores it in its repository for other SOR tasks to use in their computation, (see Figure 3).

The test node exits the computation if the maximum number of iterations has been exceeded or the tolerance for the computation has been met. This tolerance is computed in the SOR task node by computing the value $\| Ax - b \|$. Note that this is a global convergence test.

This computation is nondeterministic. The parallel SOR tasks write to the local copies of their x repository while other SOR tasks running in parallel read from these repositories, (see Figure 4).

It is possible to alter this computation slightly so that it is deterministic, (see Figure 6). In the deterministic SOR, each iteration of the computation is carried out in a synchronized fashion. After new values for x are computed, all parallel processes synchronize and the x vector is updated in a barrier fashion. The code for **start** and **sor** will need to be altered slightly from Figures 2 and 3 to reflect the differences in repository access for these nodes in the deterministic SOR of Figure 6. For the SOR computation, the nondeterminacy may be tolerated since the computation will converge even if the x values are updated asynchronously. Therefore, determinacy may be traded for efficiency in this case.

4 Adaptive Global Optimization

This algorithm is an adaptive parallel iterative stochastic method for finding the minimizers of a function in some space S [12]. In the parallel version, S is subdivided into regions which are operated upon concurrently. For each subregion at each iteration k the following steps are performed:

1. Generate a set of sample points and calculate their function values.
2. Select the start points from the sample.
3. Conduct local minimizations from each of the start points.
4. Determine whether to stop, and if not, repeat these steps.

In the adaptive version of this algorithm regions may be further subdivided, thus focusing on areas in which more work is to be done.

The Phred program for adaptive global optimization can be found in Figure 7. The main features of this program are a pipe structure, a loop, and a dynamic conditional statement.

[1]To understand the C code in Figures 2, 3, and 4 it may help to read Appendix A which describes the Phred/C interface.

```
struct mytok {
    int index;
    double tol, x;
    };

start(t,r)
struct tokio *t;
struct repio *r;
{

    struct mytok *token;
    double *a,*b;

    /* get the token pointer */
    token = (struct mytok *)t->indata[0];
    /* set the number of iterations */
    token->index = MAXITERATIONS;

    /* allocate the repositories */
    a = (double *)xmalloc(MAXM*MAXN);
    b = (double *)xmalloc(MAXN);
    r->outdata[A] = (char *)a;
    r->outdata[B] = (char *)b;

    /* read the data for a and b */
    readIn(a,b,MAXM,MAXN);

    /* create the tokens and the tasks */
    t->num_out = MAXM;
    create_tokens(t);

    /* the elts of t->outdata[*] point to
       t->indata[0] so we're set */

}
```

Figure 2: The start function code for the parallel SOR program.

```
sor(t,r)
struct tokio *t;
struct repio *r;
{
    external int myindex(); /* my sibling number */
    int i,j;
    double *my_x, *b, alpha, beta, xo;
    double omega = 1.2;
    struct mytok *token;

    struct dooble {
        double a[MAXM][MAXN];
        } *a;

    /* get a pointer to the incoming token */
    token = (struct mytok *)t->indata[0];

    my_x = (double *)r->indata[X][my_index()];

    /* use the INITIAL_X the first time around */
    if (token->index == MAXITERATIONS) *my_x = INITIAL_X;

    xo = *my_x;     /* save old x value */

    a = (struct dooble *)r->indata[A];
    /* get a handle on the A array and the B vector
       from the repositories */
    b = (double *)r->indata[B];

    alpha = 0.0;
    for (j = 0; j < my_index(); j++) {
        alpha += a->a[my_index()][j] *
                 (*(double *)(r->indata[X][j]));
        }
    beta = 0.0;
    for (j = my_index()+1; j < MAXN; j++) {
        beta += a->a[my_index()][j] *
                (*(double *)(r->indata[X][j]));
        }
    *my_x = (1.0 - omega) * xo +
            (omega * (-alpha - beta + b[my_index()]))/
             a->a[my_index()][my_index()];

    /* Calculate the tolerence every K steps */
    if (token->index % K == 0)
        /* set the tolerence to || Ax - b || */
        token->tol = calc_tol(r->indata);

}
```

Figure 3: The sor function code for the parallel SOR program.

```
test(t,r)
struct tokio *t;
struct repio *r;
{

    struct mytok *token;

    token = (struct mytok *)t->indata[0];

    /* if the loop count has gone to 0 or
       the tolerence has been met */
    if (token->index == 0 || token->tol < TOL) {
        t->arc = EXIT;    /* then exit */
        }
    else {
        --(token->index); /* decrement index */
        t->arc = CONT;    /* and continue    */
        }
}
```

Figure 4: The test function code for the parallel SOR program.

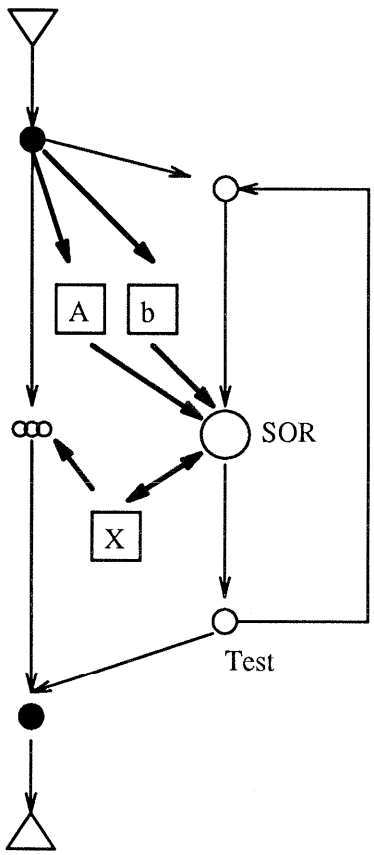


Figure 5: The Phred graph for the chaotic SOR program

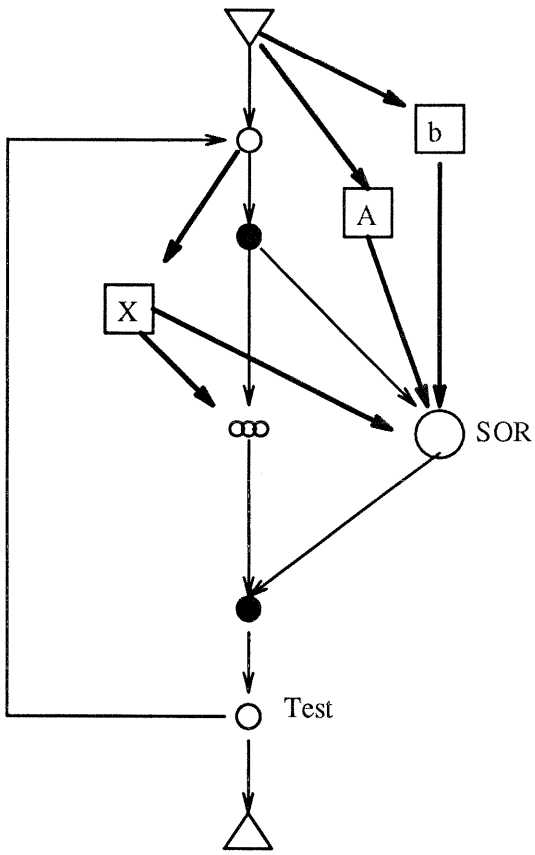


Figure 6: The Phred graph for the deterministic SOR program

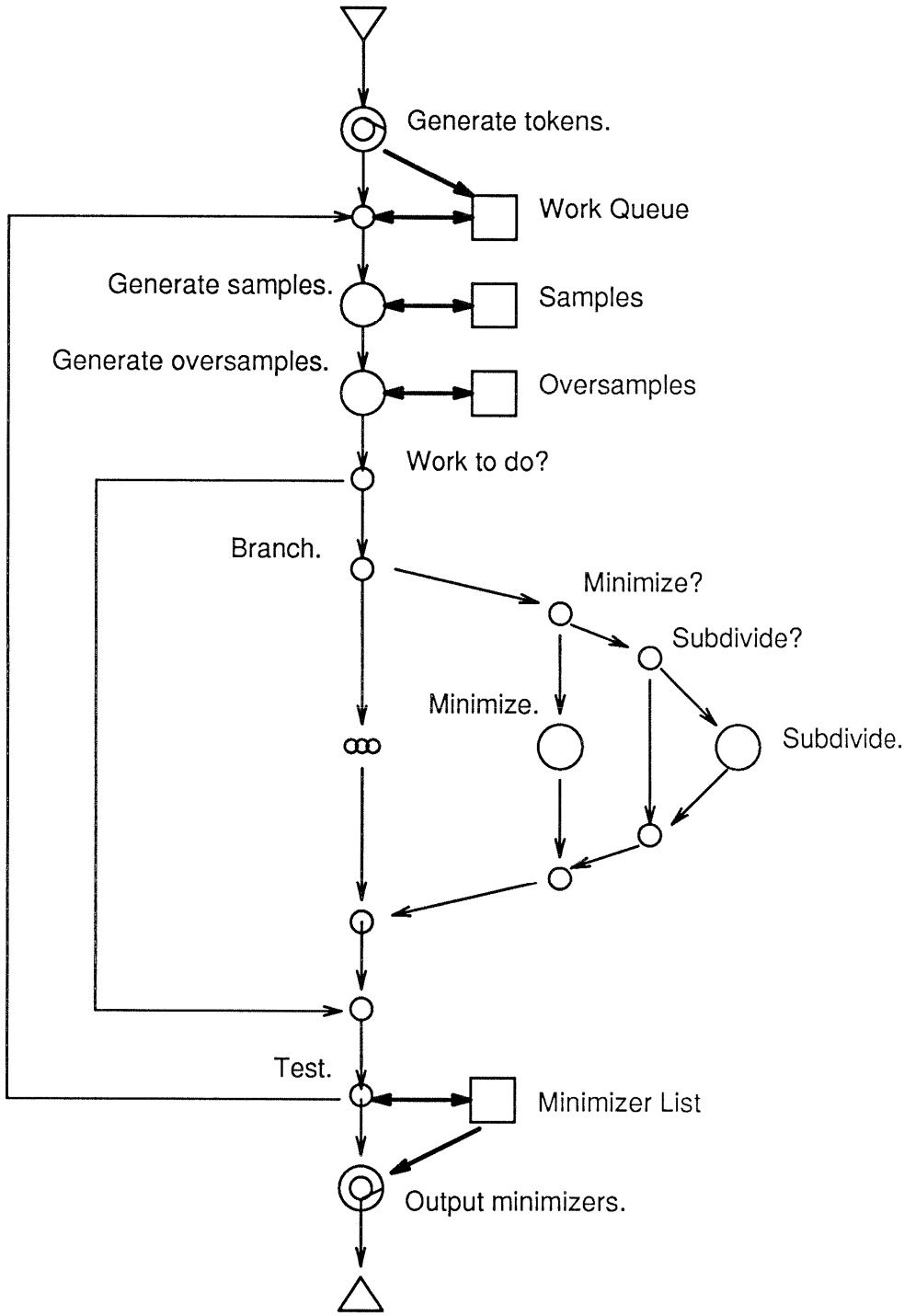


Figure 7: Adaptive global optimization program.

The pipe structure is bracketed by the **Generate tokens** and **Output minimizers** nodes. The **Generate Tokens** node initializes the work queue data repository with the regions to be optimized and emits several tokens, each representing a parallel process involved in the optimization. These tokens then proceed into the main loop of the program where they execute nodes of the main loop in parallel. In Phred the data structures for the work queue, the sample points, the over sample points, and the minimizer list are stored in data repositories as part of the data flow graph. When a region is subdivided, the newly created region is placed in a work queue. Sample and over sample points are shared among the parallel processes. Thus each token removes a region from the work queue. Note that only one token at a time may access the work queue data repository since control flow nodes in Phred are executed atomically. That is, single nodes within a pipe structure act as critical sections. However while one token may be executing the task node, others may be executing different nodes in the subgraph between the **Generate tokens** and **Output minimizers** nodes.

After a token receives a region from the work queue, it continues to the **Samples** and **Oversamples** nodes where related samples are read from the data repository and any newly computed samples are stored. A token carries with it relevant sample and oversample points. Oversample points are points falling on the border between subregions. They, unlike other sample points, may be utilized by more than one subregion. This corresponds to Step 1 of the algorithm as described previously. The data structures for storing the samples are complex because of the dynamic nature of the algorithm. Access to the sample data structures occur in a critical section, yet other tasks may be executing different parts of the program in parallel which do not access these data repositories. At this stage the region as well as any relevant sample and oversample information is carried along in the token.

The **Branch** node is like the begining of a large case statement. It's purpose is to channel each token down a separate but identical subgraph. The first token to arrive at the **Branch** node will make a Phred system call to create N subgraphs, one for each token in the pipe structure. After the subgraphs have been completed the tokens may continue toward the separate subgraphs based on a unique token number stored in each token. This allows the different subgraphs to be executed in parallel while loosely synchronized by the flow of tokens.

If a region has not been minimized, the token flows through the **Minimize?** node to the **Minimize.** node. Here, start points for the local minimization are generated, Step 2. At the **Subdivide?** node in the replicated subgraph each token decides whether its region should be further subdivided, if so the token takes the path to the **Subdivide** node. The newly created regions and any sample points will then be carried by the token back to the top of the loop and stored in the appropriate repositories. The token will obtain another region to compute and continue. Once the local minimizers have been computed by the **Minimize** node for Step 3 of the algorithm, they are carried to the bottom of the loop where they are stored. Any new sample points that were computed during the minimization are placed in the correct repositories at the head of the loop as the token returns for more work. The **Test** node decides whether to stop, Step 4. If there is no more work to be found the token exits the loop and waits to be absorbed by the end pipe node labeled **Output Minimizers**.

According to the Phred semantics, a pipe end node does not execute until all of the tokens which were spawned by the corresponding pipe begin node have arrived. Thus the **Output Minimizers** node will only execute after all of the tokens have placed their minimizers in the minimizer list

and checked that the work queue is empty. The minimizer list will be output and the program terminates.

5 Conclusion

In related work, we used ParaDiGM to model parallel numerical algorithms [5, 12]. In this paper we have shown how complex parallel numerical algorithms can be implemented in Phred. The examples presented here were both static and dynamic. The most of the low level details of the algorithms are appropriately hidded in the Phred representation while the essential elements of parallelism, data flow, and control flow are expressed visually at a high level. While there are certainly types of parallel computations which cannot be expressed in Phred, it should be apparent that a large number of interesting parallel computations can be expressed in Phred. The expressiveness of Phred, coupled with the ability to automatically analyze Phred programs, makes Phred particularly suited to exploring the role of determinacy in parallel programming.

6 Acknowledgements

We would like to thank Sharon Smith for relentless proofreading and insightful comments, without which, Section 4 of this paper would probably not exist. The authors also thank NSF for supporting the the research under Grant No. CCR-8802283.

References

[1] A. Beguelin and G. J. Nutt. A tool and language for visual distributed programming, December 1990. Submitted for publication.

[2] A. Beguelin and G. J. Nutt. A visual parallel programming language, October 1990. Submitted for publication.

[3] Adam Beguelin. *Deterministic Parallel Programming in Phred.* PhD thesis, University of Colorado, Department of Computer Science, Boulder, Colorado 80309-0430, 1990.

[4] Richard L. Burden and J. Douglas Faires. *Numerical Analysis.* Prindle, Webber & Schmidt, 2 edition, 1985.

[5] Isabelle Demeure, Sharon L. Smith, and Gary J. Nutt. Modeling parallel, distributed computations using paradigm — a case study: The adaptive global optimization algorithm. In J. Dongarra and P. Messina, editors, *The Fourth Siam Conference on Parallel Processing for Scientific Computing*, pages 154–161. Siam, December 1989.

[6] J. J. Dongarra and D. C. Sorensen. SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs. In D. B. Gannon L. H. Jamieson and R. J. Douglass, editors, *The Characteristics of Parallel Algorithms*, pages 363–394. The MIT Press, Cambridge, Massachusetts, 1987.

[7] L. H. Jamieson. Characterizing parallel algorithms. In D. B. Gannon L. H. Jamieson and R. J. Douglass, editors, *The Characteristics of Parallel Algorithms*, chapter 12. The MIT Press, Cambridge, Massachusetts, 1987.

[8] Gary J. Nutt. Olympus: An extensible modeling and programming system. Technical Report CU-CS-412-88, University of Colorado, Department of Computer Science, Boulder, Colorado, 80309-0430, October 1988.

[9] Gary J. Nutt, Adam Beguelin, Isabelle Demeure, Stephen Elliott, Jeff McWhirter, and Bruce Sanders. Olympus user's manual. Technical Report CU-CS-382-87, University of Colorado, Department of Computer Science, Boulder, Colorado, 80309-0430, June 1989.

[10] M. Roberts and P. M. Samwell. A visual programming system for the development of parallel software. In *2nd International Conference on Software Engineering for Real Time Systems*, Cirencster, U.K., September 1989. IEE.

[11] Nan C. Shu. *Visual Programming.* Van Nostrand Reinhold, New York, 1988.

[12] Sharon L. Smith, Elizabeth Eskow, and Robert B. Schnabel. An adaptive, asynchronous parallel global optimization algorithm. In J. Dongarra and P. Messina, editors, *The Fourth Siam Conference on Parallel Processing for Scientific Computing*, pages 192–197. Siam, December 1989.

A C Interface to Phred

A standard interface allows the user to write code for each node in a Phred graph to access the data carried on tokens and stored in repositories. The underlying execution environment deals with sending and receiving tokens as well as reading and writing repositories. The system calls the user's function after it has received incoming tokens and read all appropriate repositories. After the user function executes, all write-connected repositories are written and outgoing tokens are sent along the appropriate outgoing arcs.

The execution environment passes two parameters to the user function, `token` and `repository`. `token` is a pointer to a `struct tok_io` and `repository` is a pointer to a `struct rep_io`. The tok_io (token I/O) structure contains information about the tokens entering and exiting the task node. The rep_io (repository I/O) structure contains information about the read and write repositories connected to a node. The definition of these structures are shown in Figure 8.

Different elements of these structures are accessed using arc numbers. The data arcs coming into a node are numbered `[0, repository->num_in)` while the data arcs leaving the node are numbered `[0, repository->num_out)`. The control arcs entering a node are labeled from `[0, token->num_in)`, where the outgoing control arcs are numbered `[0, token->num_out)`. These numbers are assigned by the Phred graphical editor and can be viewed when creating a Phred program.

A.1 The Token I/O Structure

A node may access information about the incoming tokens and place information on outgoing tokens via the tok_io structure. The `arc` field contains the arc number of the arc upon which the token entered. Upon exit, the `arc` field tells the system where to send the token. The `arc` field only makes sense for disjunctive nodes. The `num_in` field is the number of incoming control flow arcs that are connected to the node. The value of this field may be greater than 1 for converging conjunctive or pipe nodes. The `num_out` field is set to the number of outgoing control flow arcs. If the node is a dynamic diverging conjunctive node or a diverging pipe node then `num_out` will be set to -1 since the system doesn't know how many tokens will be leaving the node. In such a case, the programmer should set `num_out` to the appropriate number and call the `create_tokens()` function. This will tell the system how many tokens will be leaving the node. The `create_tokens()` call will also setup the `outdata` array of the tok_io structure to accommodate these tokens. If the node is a diverging dynamic disjunctive node then the first time a token enters that node the `num_out` field is set to -1 and the user function will have to set `num_out` and call `create_tokens()`. Subsequent tokens entering the diverging dynamic disjunctive node will see the `num_out` value left by the previous token. Subsequent tokens may still alter `num_out` and call `create_tokens()`. The elements of the `token->indata` array, `token->indata[0] through token->indata[token->num_in - 1]`, point to the tokens which were read by the system. For instance `token->indata[token->arc]` points to the token which made a disjunctive node fire. The elements of the `outdata` array point to the tokens to be sent out on the various arcs; i.e., `token->outdata[2]` points to the token that will be sent out arc number 2.

Sensible defaults are defined for setting up pointers in the `indata` and `outdata` arrays for the different types of nodes. For diverging disjunctive, conjunctive, and pipe nodes, all the elements of `outdata` will point to the same location as `indata[0]`, the single input token. This means that for a node of this type, if the tok_io structure is not modified, copies of the incoming token will automatically be dispatched along all the appropriate out arcs. For converging disjunctive nodes the single `outdata` pointer, `outdata[0]`, will point to the same location as `indata[arc]`. Thus the incoming token will automatically be routed through the node. For converging conjunctive and pipe nodes the single `outdata` pointer, `outdata[0]`, will point to the same location as `indata[0]`. This is an arbitrary choice; one of the incoming tokens is simply passed on. For tasks nodes, the `outdata[0]` element will point to the same location as `indata[0]`.

There are restrictions as to how the `outdata` array may be used. For instance, all nodes except the stop node must emit at least one token. The `outdata` array must be consistent with the definition of Phred as described in [3].

In order for the system to know the length of the token data structures which will be passed on, the pointers stored in the `outdata` array must either be those that were in the `indata` or `outdata` arrays when the function was entered, or those allocated using the `xmalloc()` function call. The `xmalloc()` call returns a pointer to a contiguous piece of memory and also keeps track of the length of this memory. This way the system may free this memory after the token has been sent on to the next node in the program. The programmer need not and should not free the data pointed to by the `indata` or `outdata` array elements; the system will automatically free this memory.

A.2 The Repository I/O Structure

The rep_io structure is to repositories as the tok_io structure is to tokens. A node may access information about the read- and write-connected repositories via the rep_io structure. The num_in field is set to the number of read repositories connected to the node. If there is a local repository connected, it is counted only once in num_in. Tasks may find out how many local copies of a repository were spawned by calling the num_siblings() function. The num_out field is set to the number of write repositories connected to the node. If there is a local repository, it is counted only once in num_out. Neither num_in nor num_out should be altered by the programmer. The indata array elements point to the repository data that was read before the function was called. If there is a local repository which is also accessed by a node's siblings, then the element of indata for that repository will point to an array of pointers to the local repositories for each sibling task, not directly to a repository. If no other siblings reference a local repository, then there is only one local repository for the node to access and it is directly pointed to by an indata element. The number of local repositories is equal to the number of tasks spawned; this can be determined by calling num_siblings(). For instance, indata[DYN][num_siblings()-1] points to the data from the repository allocated with the last spawned task. Furthermore, indata[DYN][my_index()] is the node's own local data repository. (Note that DYN is the arc number for the local repository.)

The elements of the outdata array point to the repository data to be written upon completion of the function. If a pointer in this array is NULL then the repository will not be written. If the repository is both write- and read-connected to a task, then the indata and outdata pointers to that repository will point to the same location. (Note that the indexes into the indata and outdata arrays may be different since they are based on arc numbers.) If an element of outdata represents a local repository that is accessed by siblings, then this element will point to an array of pointers to repositories, not to a single repository. All other pointers in the outdata array will be set to NULL. If the programmer wishes to write to these repositories then he must either xmalloc() storage for this repository and place the pointer in the correct element of the outdata array or assign an element of the indata array to the correct element of the outdata array so it points to storage that already exists. The pointers stored in outdata must be either those that were in indata or outdata when the function was entered, or those allocated using xmalloc(). This way the system knows the length of the data to send. As in the token case, the repository data pointed to by the elements of the indata and outdata arrays will automatically be freed by the system after the repositories are written. The programmer should not free these pointers.

A concrete example may shed more light upon the repository interface. Figure 9 shows a section of a Phred program graph for a task in a dynamic conjunctive structure that accesses a local repository. The indata and outdata arrays in the rep_io structure are also shown. Each task T is spawned along with a copy of the repository R. T is only able to write to its copy of R, $R_{my\_index()}$, but can read from all copies of R, $R_0 \ldots R_{num\_siblings()-1}$. If T read a shared repository then indata[S] would directly point to a copy of this repository. (Here S is the arc number for the shared repository.) If T read another local repository, then indata[L] would point to an array of pointers to the copies of the local repositories similar to indata[0]. Here again L is the arc number for the local repository.

The C interface presented here allows the programmer to manipulate token and repository

data in the Phred programming environment. The programmer simply writes code for the node interpretation routines. The run-time system will call a function when its node is activated. The repository and token information is passed to the routine through the parameter list. Outgoing token and repository information is also passed out through the same data structures.

```
struct tok_io {
    int arc;
    int num_in;
    int num_out;
    char *indata[];
    char *outdata[];
    };

struct rep_io {
    int num_in;
    int num_out;
    char *indata[];
    char *outdata[];
    };

extern int my_index();

extern int num_siblings();

extern int create_tokens(struct tok_io *token);
```

Figure 8: C interface structures

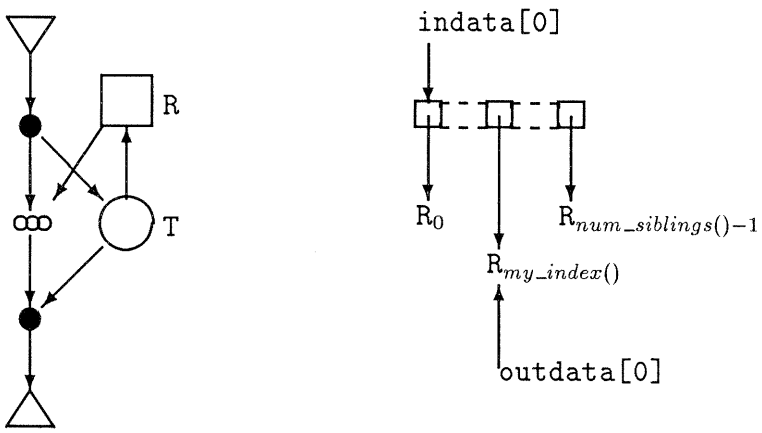


Figure 9: Repository access data structures.

A Tool and Language for Visual Distributed Programming

Adam L. Beguelin†
University of Tennessee - Knoxville
and
Oak Ridge National Laboratories

Gary J. Nutt‡
University of Colorado (USA)

---

† This work was done while this author was at the University of Colorado, supported by NSF Grant CCR-8802283. His current address is Department of Computer Science, Ayers Hall, Room 104, University of Tennessee, Knoxville, TN 37996.

‡ This author was supported by NSF Grant CCR-8802283. His address is Department of Computer Science, Campus Box 430, University of Colorado, Boulder, Colorado, 80309-0430 USA. Telephone: (303) 492-7581; FAX: (303) 492-2844. Electronic mail address: nutt@boulder.colorado.edu.

ABSTRACT

Software technology continues to lag behind distributed computer hardware technology, particularly for application software. Although significant progress has been made in vectorization and parallelization of sequential programs for specialized parallel machines, there has been substantially less progress for general distributed computer systems. As a result, many of today's network systems are essentially used as collections of individual serial computing resources.

This paper describes an experimental language, *Phred*, and an associated tool designed to make it easier to program distributed computer systems, particularly by applications programmers. The language is based on a formal graph model of computation, constrained so that portions of the program that allow nondeterminacy can be identified with a static analyzer. The tool provides facilities to construct and analyze a program in the language, and to test it by interpreting the underlying graph model.

The language and tool focus on two issues: assisting application programmers to create software that contains explicit parallel constructs, and helping to identify software that may not behave deterministically. The first issue is addressed by designing the language and environment explicitly to represent distributed computation models visually. The second issue is addressed by the ability of the support environment to recognize program constructs that may result in nondeterministic execution. The synergy of language and tool yields a programming environment that takes advantage of conventional language features and contemporary interactive computing facilities.

The paper outlines the language, illustrates the use of the language and the static analysis through simple examples, then describes the architecture of the tool. An interesting aspect of the tool is the means by which the distributed architecture is used to implement a nonintrusive syntax-directed graph editor for the language.

1. INTRODUCTION

Software technology continues to lag behind distributed computer hardware technology, particularly for application software. Significant progress has been made in vectorization and parallelization of sequential programs for specialized parallel machines, e.g., vector and pipelined machines, but there has been substantially less progress for MIMD machines and distributed computer systems. As a result, many of today's network systems are essentially used as collections of individual serial computing resources.

Our overall goal is to explore languages and tools to make better use of parallel and distributed computer systems, particularly by applications programmers. To that end, we describe an experimental programming language, *Phred*, for parallel computer systems and an accompanying set of tools for constructing programs in the language. The synergy of language and tool yields a programming environment that takes advantage of conventional language features and contemporary interactive computing facilities.

In this study, we concentrate on two aspects of the problem of constructing software for distributed systems: the first aspect is in analyzing the distributed program to determine if it is guaranteed to be functional -- or if it is not, then to identify the parts of the program that potentially cause the nonfunctional behavior; and the second aspect is related to selecting intuitive mechanisms for expressing the parallelism in the computation,

We have designed the Phred programming language and support environment to allow a software designer to create Phred programs, to statically analyze them for determinacy, and to interpretively execute them. Programs written in the language can also be translated into conventional procedural source programs with corresponding system calls to support parallel operation. In this paper, we describe the language, the analysis mechanism, and the interpretation system. The approach is described in more detail in Beguelin's thesis [7].

1.1. Remarks on Distributed Programming

Distributed software development differs from sequential software development in that the programmer must control the creation, destruction, and interaction of parallel units of computation. The intellectual activity required to manage this complexity is substantially greater than the activity for controlling a single instruction flow in sequential programs. In related modeling work, we have addressed several of these issues related to process definition and control [10]. Phred focuses on the issue of managing threads of execution that share information.

Determinacy

A set of individual units of computation are known to be *determinate* if and only if they are mutually noninterfering, i.e., there is precedence among the units if they violate Bernstein's conditions, (see Chapter 3 of [26]). Thus, to decide if a distributed computation is determinate, it is necessary to:

- identify all of the individual units of computation that can exist at any given time
- determine if they violate Bernstein's conditions
- if they do, determine if they are allowed to be executed simultaneously

Determinacy analysis can be facilitated by the language construction, or alternatively made very difficult by the language and run time environment. For instance, it is not difficult to detect blocks of code that can be executed in parallel in a program written in a functional programming language. Conversly, if the distributed program is written in a conventional language such as C with underlying operating systems facilities for process management (e.g,. UNIX **fork** and **wait**), then there is little chance that static analysis techniques can be used to identify the set of processes that exist at any given instant.

When a programmer accustomed to writing serial programs begins to construct distributed programs, he will then be exposed to a wide variety of new kinds of bugs -- timing-dependent ones -- unless the programming environment accommodates him by detecting cases where the program allows nondeterminate execution to occur. Our experience with distributed and parallel programs suggests that this family of bugs will be far more difficult to deal with than, say, pointer arithmetic bugs of the type that are prevalent in serial C programs.

We believe that a useful parallel programming environment ought to be able to detect situations in which the overall computation may be nondeterministic; however, the environment should not *require* that all programs be determinate, since there are cases when that is the intent of the program designer, typically for reasons of efficiency. It is the inadvertent existence of potential nondeterministic operation that needs to be called to the programmer's attention.

Visualization of the Program

One way to construct a programming language that supports static analysis for determinacy is to construct it on top of some form of precedence graph. (This follows since it is easy to find necessary and sufficient conditions for determinacy by using the precedence and data flow graphs.) The precedence graph must be extended from traditional directed, acyclic graphs since it must be able to represent cycles (corresponding to loops in procedural languages) and since it must also represent both alternation and parallelism.

We have embraced this approach, and have been influenced by the early work of Estrin and his students in choosing formal graph models to represent parallel computation (Baer surveys that early work in [3]). Estrin's Graph Model of Behavior (GMB) employs nodes with OR and AND to represent alternative and concurrent control flow. We also employ the idea of tokens (used in Petri nets [21]) as a fundamental notion for representing precedence and concurrent activity in the graph model.

Phred is built upon a formal graph model that uses OR and AND logic, and which identifies the state of execution through the distribution of tokens throughout the graph. Phred also borrows from dataflow models [1, 9] to allow each token to have an associated list of attributes.

The paper by Bal, Steiner, and Tanenbaum provides a comprehensive survey of the spectrum of programming languages for distributed computing systems [4]. Formal graph models that describe control flow in parallel and distributed programs, might also serve as the basis of visual distributed programming languages. This is accomplished by mapping the underlying formal model into an intuitive visual model. Estrin and his coworkers employed these ideas in SARA [12]; various other researchers have also used the approach in their own visual programming languages, e.g., CAPS [18, 19], CODE [8], colored Petri nets [17], Gilt [27], LGDF [2], Pegasys [20], Predicate/Transition nets [14], Quinault [22], Raddle/Verdi [15], and STATEMATE [16].

The Phred study differentiates itself from the other visual programming languages by first focusing on the formal model that will allow the programs to be statically analyzed for determinacy, then by using that formal model as the basis of a visual programming language.

Like many other visual programming systems, we also take advantage of the computing environment to provide tools to support the programmer while he develops parallel and distributed software. We do not advocate a full software methodology in this paper, but we do argue that it should be possible to provide a set of tools that are well-matched to the design of the programming language. In our case, these tools provide:

- a graphical interface for specifying Phred program graphs
- static analysis tools
- an interpretive execution environment
- execution environments for specific distributed hardware configurations
- a language translator (source program generator)

While we have built tools of the first three types, we have not yet produced tools of the latter two types. An added benefit of our graphical interface and static analysis tool is a unique design for a non-intrusive *syntax-directed graph editor* which we describe in some detail later in the paper.

In the remainder of this paper we review the language definition, then describe the design of the tools, emphasizing their intimate relationship with the language design.

2. THE PROGRAMMING LANGUAGE

A Phred program is a marked control flow graph, a data flow graph, and a set of node interpretations. A computation is expressed in Phred by defining the two graphs (using a mouse-based graph editor), then by annotating the nodes in the graph with node interpretations. The program can be interpreted by supplying structured data records and by starting the interpreter. The graph semantics specify when nodes can be interpreted, and the resulting state transition resulting from the interpretation.

2.1. The Language Primitives

The Phred control and data flow graphs are made up of a collection of nodes of the type shown in Figure 1 (control flow graphs do not include **repository** nodes, but data flow graphs include all node types shown in the figure). The syntax for interconnecting the node types will be explained using examples that appear later in this section.

Phred graphs are single-entry-single-exit graphs, distinguished by the **entry** and **exit** nodes shown in the figure. Basic computation is represented by the **task** node, while the **AND**, **OR**, **pipe-begin**, **pipe-end**, and **ellipsis** nodes are used to describe various control flow semantics.

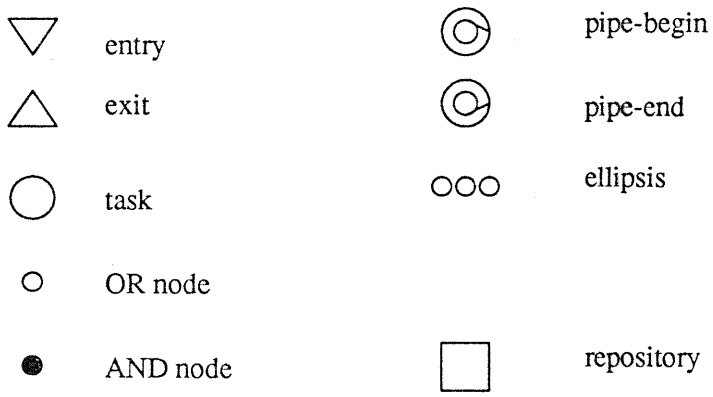


Figure 1: Phred Components

Any node in the control flow graph can have an associated *interpretation*, i.e., a procedural specification of serial computation associated with the node, (cf. SCHEDULE [11]). Generally, one would expect a **task** node to have nontrivial procedural interpretations, while the others may have interpretations that are concerned with data-driven control decisions. Phred assumes only that the interpretation is supplied by a procedure which relies on a fixed parameter list; our implementations use C source programs for node interpretations.

Tokens represent the state of the computation, and the flow of control among components by dynamically marking nodes and edges within the control flow graph. If a token resides on a **task** node, then this represents the fact that a process is logically executing the corresponding node interpretation. Thus the flow of tokens through the graph represents control flow. Tokens also carry structured data from node to node through the control flow graph.† When a token causes a node to be logically executed by arriving at the node, the node procedure is invoked by passing the data contents of the token to the procedure as a parameter. Parallelism in the control flow graph execution -- multiple tokens residing on multiple **task** nodes -- is reflected by simultaneous execution of the procedures. This simultaneity could result from AND logic in the graph, or from the a pipeline of tokens entering the graph.

A data **repository** is a data storage unit which may be accessed by the procedures in control flow nodes that are interconnected to the **repository** in the data flow graph, i.e., data sharing between procedures is explicitly represented by the data flow graph. Data **repositories** can be viewed as external variables whose scope is explicitly delimited by the arcs in the data flow graph.

Data **repositories** may be read or written arbitrarily while a function executes, assuming the repository access is consistent with the connections of the program's dataflow graph. In addition, there are two subtypes of **repositories**, shared and local: shared **repositories** are persistent and may be accessed from any node in the program, but local **repositories** can only be referenced by nodes within a syntactically-defined subset. It is essential to have well-defined access patterns for data **repositories** to analyze the Phred program for determinacy.

This brief and informal description of the formal language is an overview; a more complete discussion appears in [5], and a full, formal description appears in [7].

---

† That is, technically the control flow graph is a data flow graph, although we distinguish it from data references to **repositories** by **task** procedures.

2.2. Using Phred

In Figure 2, we specify both the control and data flow of a simple computation. The control flow part indicates that the program retrieves a record, X, from a **repository**, then spawns two threads of computation. The left thread computes a value for Y (as f(X)) if X surpasses some threshold test, then writes Y into a storage. The right thread computes a new value for Y as g(X) and writes Y into the storage. That is, the **task** nodes represent the retrieval and computation procedures while the **OR** and **AND** nodes represent pure control flow computation. The **repository** nodes represent the storage for X and Y, and the arcs to/from the **repository** nodes represent writing/reading operations, respectively.

Notice that in this representation of a program, it is possible for multiple processes to be executing at different **task** nodes at the same time, e.g, one process might be executing "Y <-

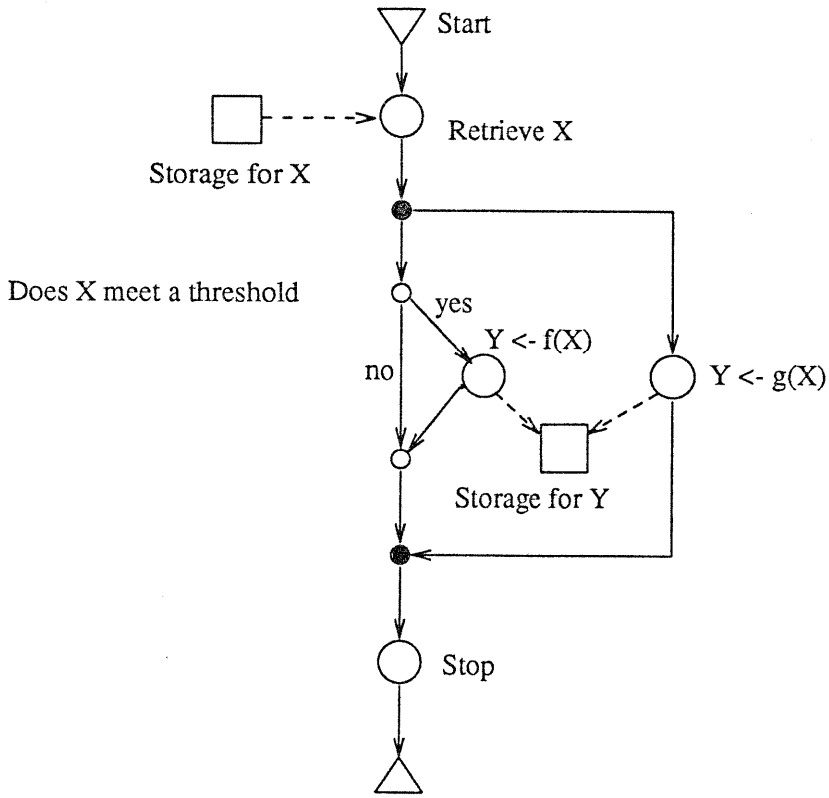


Figure 2: A Phred Example

f(X)" while another is executing "Y <- g(X)". There is the potential for nondeterminism among the two **task** nodes and the "Storage for Y" data **repository**.

The programmer might decide that this program is correct (determinate), because the units of computation are actually dealing with different records in Y; he may then proceed with development, leaving the possibility of nondeterminacy in the Phred control flow graph, but eliminating its occurrence within the procedural interpretations of "Y <- f(X)" and "Y <- g(X)". The acceptability of this approach depends upon the ultimate use of the program, in particular, how are records selected in Y -- an aspect that we have ignored to this point in our discussion.

Suppose that the procedural interpretations for the two critical units of computation appear as shown in Figure 3; the computation may result in nonfunctional behavior, since two possible values (f(x) and g(x)) are written to the repository Y in parallel.

The previous example parallelism was specified by sending multiple tokens through multiple subgraphs. In some cases we would like to send multiple tokens through the *same* subgraph. Thus we would like to explicitly identify a portion of the program as one that will behave as a pipeline, essentially overriding the precedence semantics by allowing the programmer to take responsibility for ensuring determinacy through appropriate interpretations. This is the purpose of the **pipe-begin/-end**, used as indicated in Figure 4.

A pipeline portion of a control flow graph is bracketed by the **pipe-begin/-end** node pair. When a token arrives at a **pipe-begin**, then it interprets the node's procedure; this procedure will define some number, N, of tokens to sequentially spawn on its output arcs (the default is one). The Phred firing semantics assume safety, so the stream of tokens will leave the **pipe-begin** node serially. The group of tokens will proceed through the pipe without another group entering the pipe. Multiple tokens may cause multiple task nodes within the pipe structure to fire simultaneously. Thus the normal sense of precedence is lost within a pipe structure. However, multiple tokens are not allowed to execute the same Phred node within the pipe structure. If a token arrives at a node while the node is being executed, it queues.

Figure 5 is a refinement of the previous Phred graph; it illustrates explicit parallelism through the use of the **AND** node within the pipe. Each token that enters the **AND** node named "A start" will cause M new instances of the **task** named "P details" to be created and a replica of the token to be passed to the instance. The value of M is determined by the procedure associated with "A start" (or is one in the default).

The **AND** fan-out/fan-in construct can also appear outside of a pipeline, where it is used to explicitly represent two kinds of parallelism. The first type uses the **ellipsis** (in the way it is used in Figure 5) to represent the initiation of M-way parallelism across M identical procedures. It is also possible to specify each of the M procedures statically without using the **ellipsis** node.

```
struct mytok
{
        ...
};

"Y <- f(X)"(t, r)
struct tok_io *t;
struct rep_io *r;
{
        struct mytok *token;

        ...
        // Get the value of the incoming token
        token = (struct mytok *) t->indata[0];
        ...
        // Compute and store a new value for repository Y
        r->outdata[0] = f(token);
        ...
}

"Y <- g(X)"(t, r)
struct tok_io *t;
struct rep_io *r;
{
        struct mytok *token;

        ...
        // Get the value of the incoming token
        token = (struct mytok *) t->indata[0];
        ...
        // Compute and store a new value for repository Y
        r->outdata[0] = g(token);
        ...
}
```

Figure 3: Node Procedures

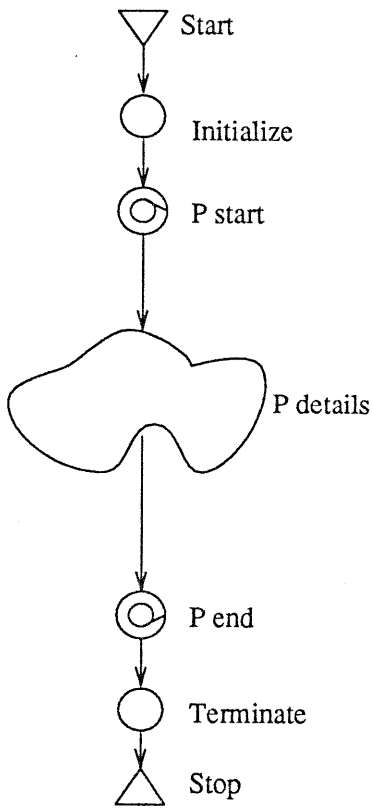


Figure 4: Phred Pipes

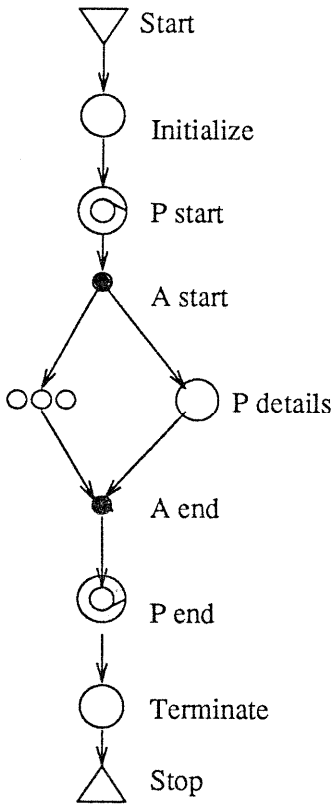


Figure 5: Phred AND Concurrency

Statically analyzing these constructs for determinacy can be challenging. Figure 6 is a refinement of the previous Phred graph, showing **repositories**. First, consider the case where a **task** outside a pipe shares a **repository** with a **task** inside a pipe, e.g., **repository** "r2" in Figure 6. In this case, the Phred semantics specify that "r2" is a single **repository** which is written by "Initialize" and read by the M instances of "P details", thus the construct is determinate with respect to access to r2. The "r3" node is a local **repository**, created when "P details" is instantiated; there is one "r3" for each "P details" instance. Even so, the manner in which the family of local "r3" instances is used can be nondeterminate since there is no order in which the instances read and write each "r3" instance.

The Phred graphs that appear in these examples are an important aspect of each program, specifying the basic control and data flow characteristics, and the concurrency in the program. It is not difficult to construct a linear language that will generate these formal models of the computation, although there is little reason to do so. Rather, we have developed a set of interactive, visual tools to support the creation of Phred programs as a combination of graphical and textual

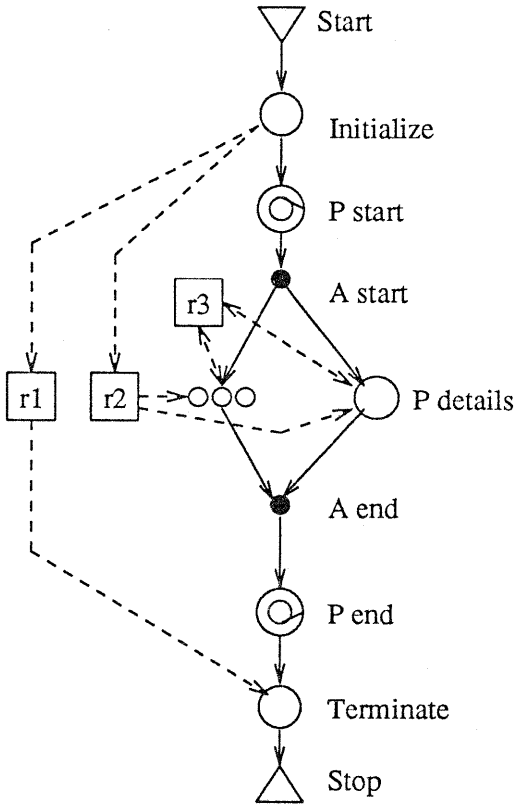


Figure 6: Repository References

information. In the remainder of the paper, we describe those tools.

This Subsection has described the character of Phred programs through simple examples. More extensive sets of examples appears in [6, 7].

3. THE PHRED SUPPORT SYSTEM

3.1. Goals of the Tool

Phred is a useful programming language for expressing distributed programs, since it inherently defines the *architecture* of the computation as well as the details of the computation. A tool has been designed and implemented to provide a visual programming facility to be used with Phred, and to analyze Phred programs as they are created.

The goals of the tools are to:

- implement an interactive Phred graph editor
- provide facilities to automatically analyze the graph for determinacy
- provide an execution environment for Phred programs

Our prototype system meets these goals by implementing the programming environment in the context of the Olympus architecture [23, 25], specifically using several aspects of the BPG-Olympus implementation [24]. By employing the Olympus architecture, we have developed a Phred tool that is itself a distributed program, and which takes advantage of the architecture to provide a flexible, interactive environment. In particular, the environment allowed us to design a unique syntax-directed Phred editor which bypasses many of the annoyances ordinarily associated with syntax-directed editors.

3.2. The Olympus Architecture

Olympus systems, such as the Phred programming environment, are implemented using a bitmap graphics user interface in which a designer constructs models or programs in the formal language supported by that system. Typically, the user employs a graphic point-and-select editor to construct a visual model; he may then provide annotation for the model through a wide variety of mechanisms, ranging from popup windows through preparing separate files in a distinct editing session.

A model (program) is exercised by using a graph interpreter to determine the status of the model, to schedule the interpretation of various nodes, and to execute the procedural **task** interpretations as specified in the model instance. The modeling system may allow the user to observe the operation of the model through the changing state of the model (distribution of tokens), or through the display of probability distributions, gauges, or other monitoring displays.

Many of today's modeling systems can be criticized on the following grounds: They may leave the user in a particular *mode* during a session, e.g., the user cannot edit if he is in the process of interpreting. It is awkward to change loading conditions while the model is in execution, since the interpretation can only be interrupted through checkpoints. Once the model is being interpreted, any editing change to the model requires that the model be halted and the editor be started (a mode transition) in order to accomplish the change. The modeling language is specific to the system rather than to the designer. Only a single user can interact with any particular model at any given time. These restrictions are overcome in the Olympus architecture.

There are no modes in an Olympus modeling session; the user is allowed to interact with any of the facilities (including the editor) at any time -- including during interpretation. The interpreter can be suspended or halted at any time, then restarted or resumed later. Multiple

users can be involved in an individual modeling session.

Olympus systems are designed as a collection of modules that interoperate using message-passing protocols, see Figure 7. The *frontend* implement the human-machine interface -- using workstations and bitmap graphics as well as conventional character-oriented terminal interfaces. The *backend* implements the logic of the modeling language, independent of the user interface implementations. *Node interpreters* execute procedural declarations associated with **task** nodes in the model.

The frontend and the backend communicate over a general network communication protocol using a specialized Olympus protocol. The protocol can be easily explained through a specific example: suppose the editor in the frontend intends to define a node in the model (in behalf of the user interacting with the editor). Then the editor uses the protocol to send a message to the backend, specifying that a node should be added to the model. The backend performs the action, then acknowledges the request by responding with a similar message to the frontend indicating that a node should be added to the model. This causes the editor portion of

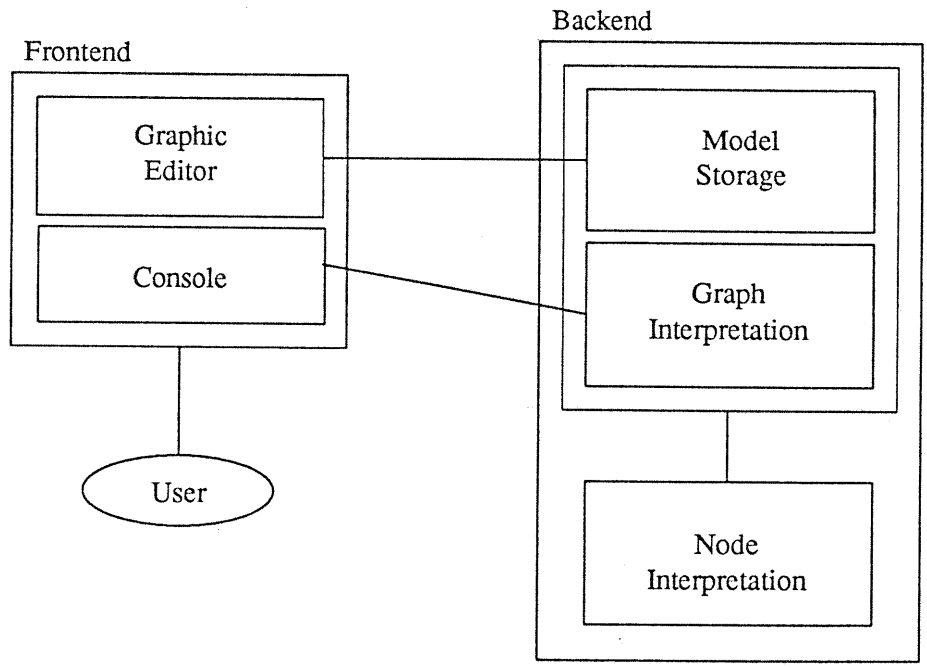


Figure 7: The Olympus Architecture

the frontend to physically display the node. That is, each operation by the frontend is acknowledged by the backend, once the backend has completed its side of the operation (storage, in this case).

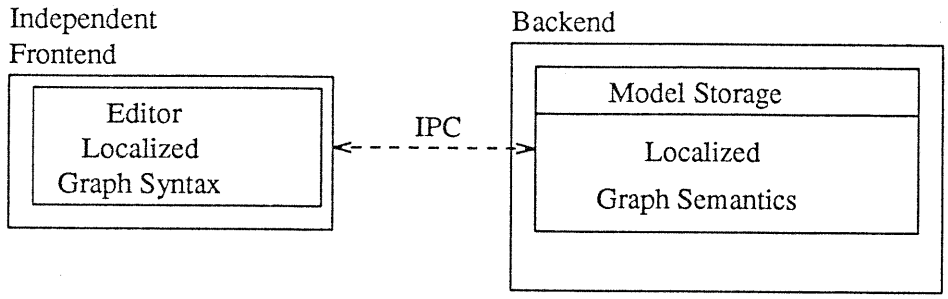
There are several interesting properties from this design: first, the backend keeps an internal representation of the model created by the frontend that is independent of the presentation used by the frontend (Figure 8(a)). For example, the frontend need not even be a graphical frontend if that is not desirable. It may also use an arbitrary user interface and visual representation scheme. The logical interpretation and the presentation of the model and its execution are separated into model syntax and semantics. The model syntax -- the appearance of the model at a workstation screen -- is implemented wholly within the frontend, while the the model's semantics -- the logical behavior of the model -- is implemented wholly within the backend.

Secondly, the frontend operation is asynchronous with respect to the operation of the backend (Figure 8(b)). This allows one to construct a frontend that is independent of the mode in which the backend operates. Editing operations can be performed at any time, resulting in a message being sent to the (storage portion of the) backend. An Olympus instance can be designed to support modeless operation at the user interface, avoiding the need for checkpoints or other mechanisms that are required in a single-process design.

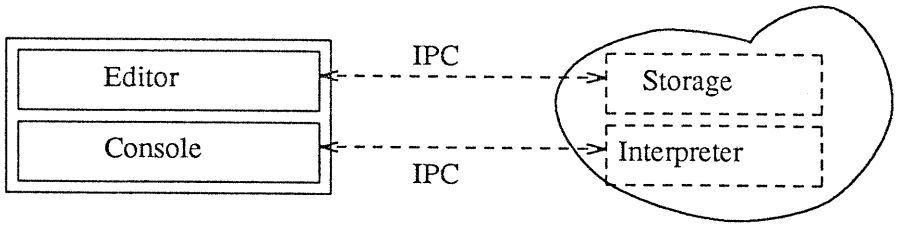
Third, the backend makes few assumptions about the state and operation of the frontend (Figure 8(c)). While it is convenient to describe the frontend process as we have done above, the backend makes no assumption about the ultimate functionality implemented in a frontend. For example, one frontend may simply send storage commands to the backend, ignoring all acknowledgements; another may send a limited number of commands to the backend, e.g., console commands, yet act upon all commands that are sent by the backend.

Fourth, the backend can support multiple frontends at any one time (Figure 8(d)). An Olympus configuration may have two or more frontends connected to a single backend. For example, two identical frontends (running on distinct machines) might be connected to a single, remote backend. Now each editing command by either frontend is echoed to both frontends, allowing each frontend to represent the current state of the model stored in the backend. (This is the fundamental aspect of the multiuser support provided by Olympus.)

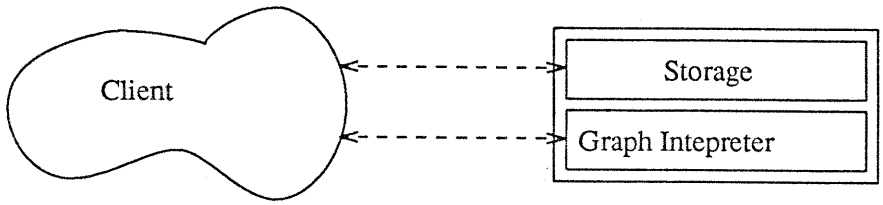
While we have strongly suggested that Olympus, itself, is a distributed system, the nature of the distribution is determined by a particular instance of the architecture. The BPG-Olympus instance is implemented in a Sun workstation environment. The frontend and backend are interconnected using the BSD socket facility, allowing very general location of the frontend and the backend processes.



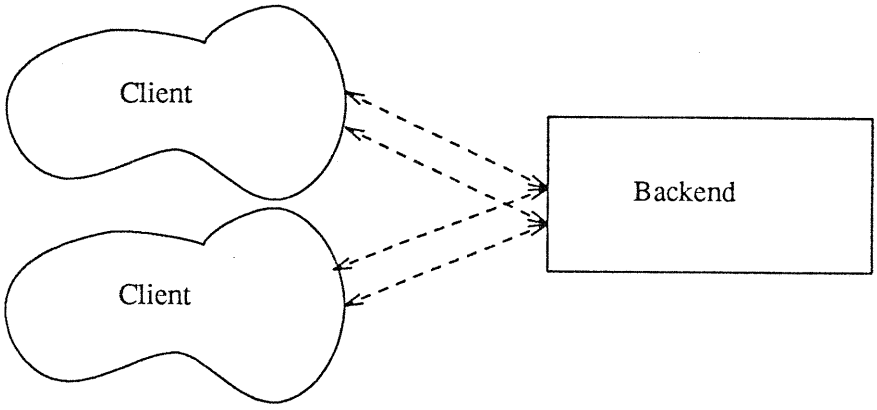
(a) Independence of Semantics from Syntax



(b) Asynchronous Operation



(c) Backend Makes Few Assumptions about the Frontend



(d) Multiple Frontend Clients

Figure 8: Views of the Olympus Architecture

There are several different frontends, some implemented in the SunView window system, others in the Sun NeWS window system. (We have also implemented a "teletype" frontend and another frontend in a Symbolics workstation.) A NeWS frontend is actually implemented as at least two processes, one for the NeWS server and another to implement the frontend application.

The BPG-Olympus backend is implemented as several processes, each potentially located at a distinct node in the network. The frontend interacts with a basic backend process, which implements model storage, interpretation status (marking distribution), and token movement on the graph model. Node procedures are executed by individual processes under the direction of the basic backend, using Sun RPC to coordinate their execution with the graph token movement.

3.3. The Organization of the Phred Tool

The Phred tool is an instance of the Olympus architecture, see Figure 9. The components in the tool are an editor-console frontend, a static analyzer frontend (called the *critic*, inspired by work of Fischer and his colleagues [13]), and a backend execution environment for Phred programs. The tool takes advantage of the multiple user capability of the backend by creating a second frontend process at the user's workstation. The critic process is independent of the normal editor-console frontend, and may be executed within the same window system on the editor-console machine (as indicated in the Figure), or at any site on the network.

The Editor-Console Frontend

The Editor-Console is a small variation on the Sun NeWS frontend used in BPG-Olympus. It employs the same user interface model and IPC mechanism between the frontend and backend, but the nodes in the Phred editor reflect the node set for Phred. That is, several new types of nodes unique to Phred are stored in the server as well as the usual BPG nodes. This allows Phred to be built on top of the unchanged BPG backend.

The BPG editor employs a type hierarchy representing the appearance and visual behavior of nodes. The Phred editor extends the type hierarchy to represent Phred nodes and edges.

In general, the Phred editor does not check the syntax of the graph that is constructed by the user, nor does the model storage system (it is a BPG backend). Instead, the critic frontend is designed to accept model descriptions as they are echoed back to the editor, and to construct and analyze the resulting Phred graph.

The Critic

The determinacy analysis is of fundamental importance to the utility of Phred. It is also an aspect of the language that needs to be checked constantly, much as one checks the syntax of the language. However, checking should not be obtrusive, since its utility will quickly become more

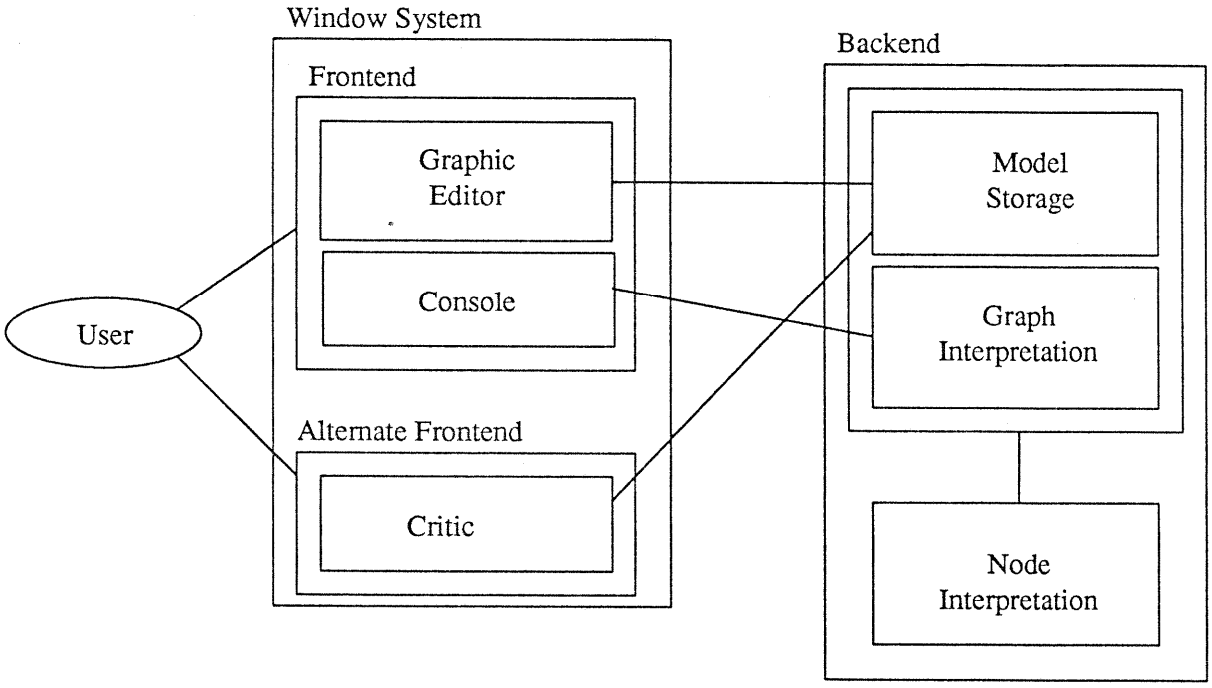


Figure 9: The Phred Tool Architecture

of a hindrance than an aide.

Syntax-directed editors for conventional programming languages have similar constraints (although the constraints are magnified in Phred due to the complexity of the analysis); while they force users to construct syntactically correct programs, they tend to be slow and annoying while the program is in an interim state of completion.

The editor incorporates a minimum amount of syntax-directed operation, e.g., if the user draws an arc between a **task** node and a data flow node, the editor will determine that the arc is a data flow arc rather than a control flow arc. However, the editor has no knowledge of determinacy, nor even of the full Phred graph syntax.

The critic is an asynchronous process that is able to analyze a formal graph to determine if it is a syntactically correct Phred graph, and to assess its determinacy properties.

An essential element of the Phred language is the ability to statically analyze the control and data flow graphs for determinacy. The critic has been designed to:

- Parse the graphs to determine if they are syntactically correct

- Analyze syntactically correct graphs for determinacy

The critic infers the graph structure as it is being created by an editor. Whenever the graph changes, the backend echoes the change to the editor and the critic. The critic begins parsing the graph (the editor is a separate process, so this parsing operation is independent of the editor -- visually and with respect to monopolizing the editor process). If, during analysis, the user changes the graph, then the update will be observed by the critic, causing it to begin parsing the graph model.

The graph parsing algorithm is a relatively straight-forward recursive descent with the exception that there is some ambiguity in the way **OR** nodes can be used in a Phred program. The parser must be able to determine if an **OR** node is part of a loop construct or a case statement. The parser analyzes the graph for dominance relationships to make this determination.

Whenever the critic has successfully parsed the graph, and when there are no pending updates to the graph from the backend, it analyzes the graph for determinacy. This analysis involves checking for mutual noninterference under both the pipeline and **AND** control flow conditions.

Both the parsing and analysis are nontrivial algorithms -- ones which would debilitate the editor or the backend had they been implemented in either place. By implementing parsing and analysis in an asynchronous frontend process, the editor and backend can be used without irritating delays for unwanted analysis. In the simple case, the critic process may be running on the same machine that implements the frontend or the backend. However, since the system is implemented on top of sockets, the critic can be running on a third machine, independent of the CPU cycles used by either the frontend or backend.

Execution Environments

The prototype Phred execution environment is the BPG-Olympus backend. The Phred editor uses the unaltered BPG-Olympus backend to store Phred graphs. Since a subset of Phred graphs are also legal BPGs, the backend can be used execute this subset of Phred. Specifically **ellipsis**, pipe, and local repository nodes are not currently supported by the prototype execution environment. Since the BPG backend makes few assumptions about the frontends, it is unaware that the frontend editor is a Phred editor instead of a BPG editor.

The backend is a two-level interpreter. At the first level, it interprets the token flow in the control flow graph. At the second level, it executes C procedures associated with each Phred node. This is accomplished by using the Sun RPC facility to bind the C procedure to the graph interpreter. We have also modified the usual RPC mechanism to allow the backend to initiate a **task** procedure when the graph interpreter determines that this should happen, and to accept the

return from the RPC as a "callback" when the C procedure returns. This allows the graph interpreter to support true concurrent procedure execution across the control flow graph.

However, the program is still interpreted at the token-graph level. Our preliminary work indicates that it is possible to construct a compiler for Phred programs that will translate the data structure stored in the backend into an equivalent procedural program (with appropriate calls to operating system concurrency support primitives).

We also suspect that a Phred program that is determinate on the basis of the graph model is equivalent to a program constructed in a functional programming languages. If we are able to show that relationship, then we can build a tool for generating functional programs from a visual frontend.

3.4. Using the Tool

The tool relies on the multiwindowing environment to take advantage of the critic for syntax-directed editing. To run the system, the user causes the server to be initiated, then starts an instance of the editor and an instance of the critic.

Figure 10 illustrates a typical screen image when the editor and critic are running, and in which the user has constructed a deterministic Phred graph. The editor window is the larger window on the left, and the critic window is the top-level window on the right. Since the windows are built in NeWS, it is easy to scale their contents. As a result, the critic window can be made to be very small, or "full sized."

Suppose that the editing session resulted in the graph being illegal, i.e., not satisfying the grammatical rules for a well-formed Phred graph. Then the critic will determine the condition and provide feedback to the user via its own window, see Figure 11. The concentric circles drawn in the critic window tell the user that the graph is not syntactically correct.

Finally, suppose that the graph constructed in the editor window was legal but possibly nondeterministic. Then the critic highlights the nodes involved in the possible nondeterminacy. When the critic is running on machine with a color display, the determinate portion of the graph is drawn in green on a black background and the questionable nodes are drawn in red. Figure 12 is clipped image of the monochrome representation of this case; the possibly nondeterminate nodes are drawn with dashed lines.

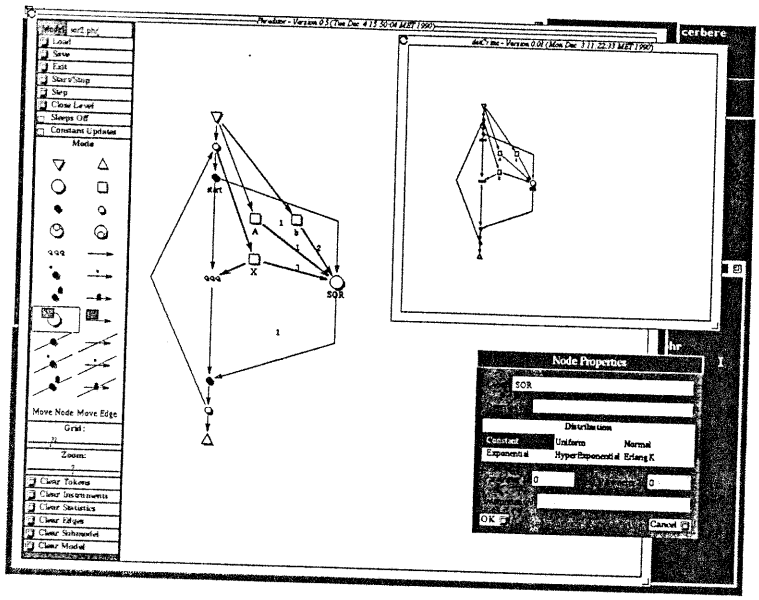


Figure 10: The Phred Editor with a Deterministic Graph

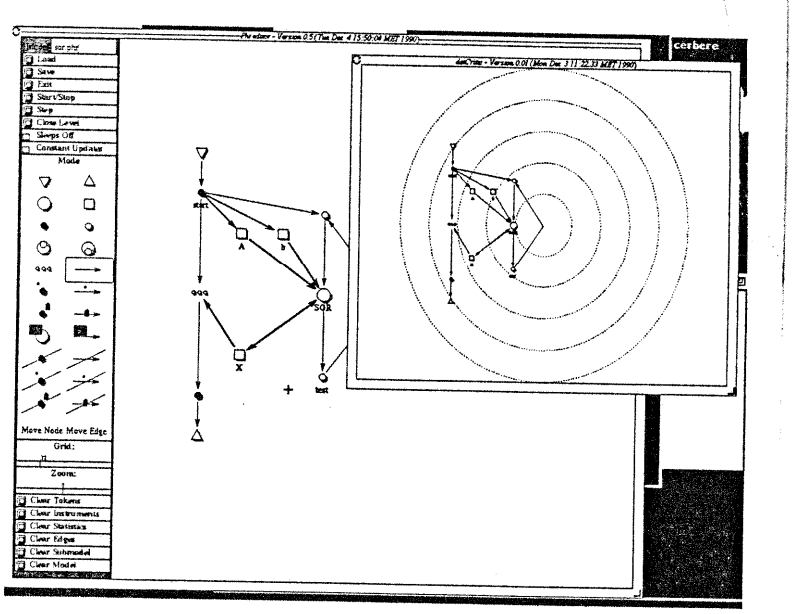


Figure 11: A Syntactically Unacceptable Graph

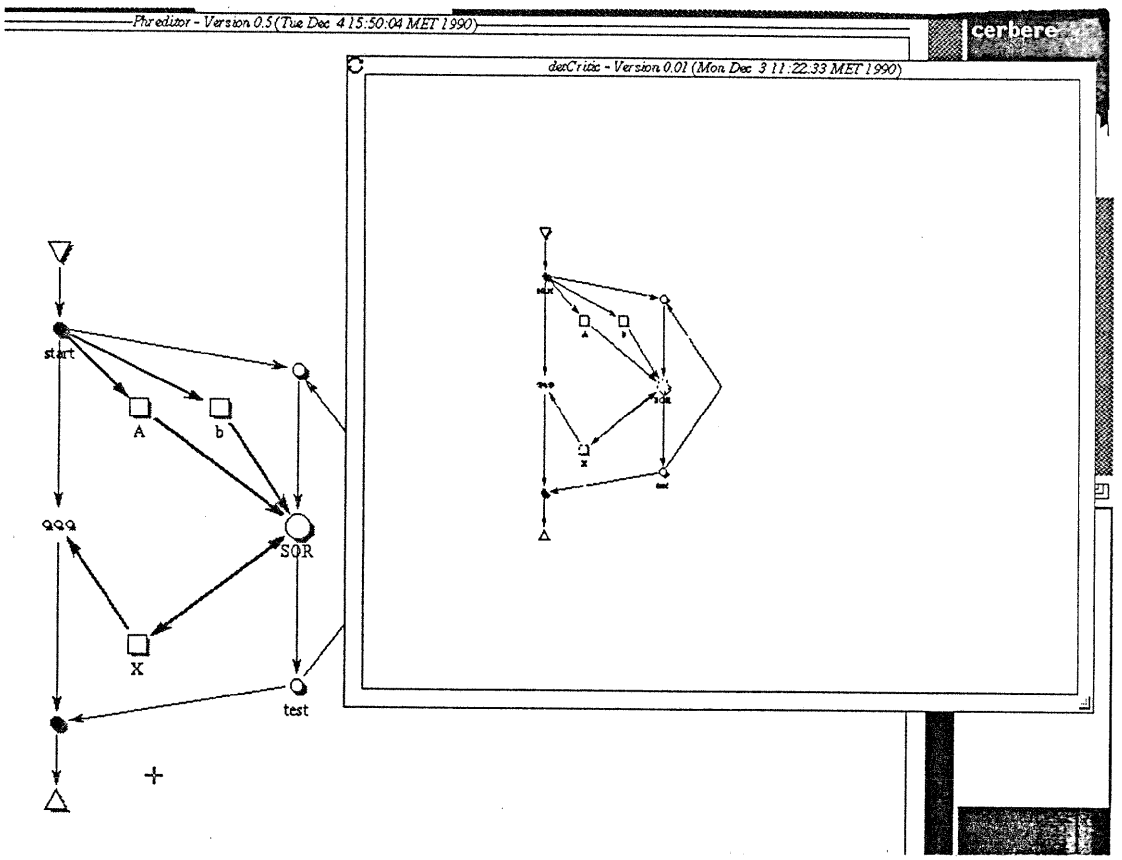


Figure 12: A Graph with Possible Nondeterminism

4. SUMMARY

Phred is a visual, parallel programming language. It has been designed to submit to static analysis for the possibility of nondeterminacy. The language does not preclude the possibility of writing nondeterminate parallel and distributed programs, it has been designed so that the distributed application programmer will be explicitly aware of the portions of the control and data flow that admit nondeterminacy.

The tool to support Phred is a visual, interactive, distributed program interpretation system. It demonstrates how formal models can be combined with a network of interactive workstations to provide a programming environment that is intuitive and potentially high performance. Our results from the research illustrate a combination of formal graph theory and distributed systems that provide a powerful paradigm for programming distributed systems.

5. REFERENCES

1. T. Agerwala and Arvind, "Data Flow Systems: Guest Editor's Introduction", *IEEE Computer 15*, 2 (Feb 1982), 10-13.
2. R. G. Babb and D. C. DiNucci, "Design and Implementation of Parallel Programs with Large-Grain Data Flow", in *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon and R. J. Douglass (editor), MIT Press, 1987, 335-349.
3. J. L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing", *ACM Computing Surveys 5*, 1 (March 1973), 31-79.
4. H. E. Bal, J. G. Steiner and A. S. Tanenbaum, "Programming Languages for Distributed Computing Systems", *ACM Computing Surveys 21*, 3 (September 1989), 261-322.
5. A. Beguelin and G. Nutt, "A Visual Parallel Programming Language", submitted for publication, October 1990.
6. A. Beguelin and G. Nutt, "Examples in Phred", submitted for publication, September 1990.
7. A. L. Beguelin, "Deterministic Parallel Programming in Phred", University of Colorado, Department of Computer Science, Ph. D. Dissertation, May 1990.
8. J. C. Browne, M. Azam and S. Sobek, "CODE: A Unified Approach to Parallel Programming", *IEEE Software 6*, 4 (July 1989), 10-18.
9. M. Broy, *Control Flow and Data Flow: Concepts of Distributed Programmings*, Springer Verlag, 1985.
10. I. M. Demeure and G. J. Nutt, "Prototyping and Simulating Parallel, Distributed Computations with VISA", submitted for publication, May 1990.
11. J. J. Dongarra and D. C. Sorensen, "SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs", in *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon and R. J. Douglass (editor), MIT Press, 1987, 363-394.
12. G. Estrin, R. S. Fenchel, R. R. Razouk and M. K. Vernon, "SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems", *IEEE Transactions on Software Engineering SE-12*, 2 (February 1986), 293-311.
13. G. Fischer, A. C. Lemke, T. Mastaglio and A. Morch, "Using Critics to Empower Users", *CHI '90 Conference Proceedings: Human Factors in Computing Systems*, Seattle, WA, April 1990.

14. H. J. Genrich, "Predicate/Transition Nets", in *Petri Nets: Control Models and Their Properties, Advances in Petri Nets 1986, Part 1*, W. Brauer, W. Reisig and G. Rozenberg (editor), Lecture Notes in Computer Science, Springer Verlag, Berlin, Heidelberg, New York, 1987.

15. M. L. Graf, "Building a Visual Designer's Environment", MCC Technical Report No. STP-318-87, October, 1987.

16. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring and M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", *IEEE Transactions on Software Engineering 16*, 4 (April 1990), 403-414.

17. K. Jensen, "Coloured Petri Nets", in *Petri Nets: Control Models and Their Properties, Advances in Petri Nets 1986, Part 1*, W. Brauer, W. Reisig and G. Rozenberg (editor), Lecture Notes in Computer Science, Springer Verlag, Berlin, Heidelberg, New York, 1986, 248-299.

18. Luqi, V. Berzins and R. T. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Transactions on Software Engineering 14*, 10 (October 1988), 1409-1423.

19. Luqi, "Software Evolution Through Rapid Prototyping", *IEEE Computer 22*, 5 (May 1989), 13-25.

20. M. Moriconi and D. F. Hare, "The PegaSys System: Pictures as Formal Documentation of Large Programs", *ACM Transactions on Programming Languages and Systems 8*, 4 (October 1986), 524-546.

21. T. Murata, "Petri Nets: Properties, Analysis and Applications", *Proceedings of the IEEE 77*, 4 (April 1989), 541-580.

22. G. J. Nutt and P. A. Ricci, "Quinault: An Office Environment Simulator", *IEEE Computer 14*, 5 (May 1981), 41-57.

23. G. J. Nutt, "A Flexible, Distributed Simulation System", *Tenth International Conference on Application and Theory of Petri Nets*, Bonn, West Germany, June 1989, 210-226.

24. G. J. Nutt, A. Beguelin, I. Demeure, S. Elliott, J. McWhirter and B. Sanders, "Olympus: An Interactive Simulation System", *1989 Winter Simulation Conference Proceedings*, Washington, D. C., December 1989, 601-611.

25. G. J. Nutt, "A Simulation System Architecture for Graph Models", in *Advances in Petri Nets*, Springer Verlag, to appear, 1990.

26. G. J. Nutt, *Centralized and Distributed Operating Systems*, Prentice Hall (in preparation), 1991.

27. M. Roberts and P. M. Samwell, "A Visual Programming System for the Development of Parallel Software", *2nd International Conference on Software Engineering for Real Time Systems*, September 1989.