Visual Parallel Programming and Determinacy: A Language Specification, an Analysis Technique, and a Programming Tool [1]

Adam Beguelin
adamb@cs.cmu.edu

Gary Nutt
nutt@cs.colorado.edu

June 7, 1993

CMU-CS-93-166

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Phred is a visual parallel programming language in which programs can be statically analyzed for deterministic behavior. This paper presents the Phred language, techniques for analyzing the language, and a programming environment which supports Phred programming. There are many methods for specifying synchronization and data sharing in parallel programs. The Phred programmer uses graph constructs for describing parallelism, synchronization and data sharing. These graphs are formally described in this paper as a graph grammar. The use of graphs in Phred provides an intuitive and visual representation for parallel computations. The inadvertent specification of nondeterministic computations is a common error in parallel programming. Phred addresses the issue of determinacy by visually indicating regions of a program where nondeterminacy may exist. This analysis and its integration into a programming environment is presented here. The Phred programming environment supports the specification, analysis and execution of Phred programs. The distribution of the programming environment itself over several workstations is also described.

[1]This paper will also appear in JPDC in 1994. This work was supported by NSF Grant CCR-8802283.

Keywords: Parallel programming, Visual programming, determinacy, race conditions.

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 rooted in the requirement for managing the activity of simultaneous threads of computation, e.g., see [7].

In this study, we concentrate on three aspects of the problem. The first aspect is related to intuitive mechanisms for perceiving the parallelism in the computation. The philosophy of our approach used is similar to that proposed by Gelernter and Carriero [16] in that we develop a *coordination language* which is orthogonal to the computation language. The second aspect is in analyzing the resulting program to determine if it is guaranteed to be deterministic and in identifying the parts of the program that contribute to potential nondeterministic behavior. Finally, we focus on tools to support the programming mechanisms and analysis techniques we have developed.

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. Phred employs the visual aspects of graph models to specify functionality, and to help control the creation, destruction, and interaction of tasks by illustrating points of synchronization and information sharing 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.

1.1 The Importance of Determinacy

Determinacy is important in parallel programming; LeBlanc and Mellor-Crummey [18] state:

> 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 designing parallel languages is spent 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.

Although correct program behavior does not necessarily imply deterministic behavior, nondeterministic programs can contain extremely subtle errors. If a programming system can notify the programmer of potential nondeterminacies, the programmer has a much better chance of avoiding the pitfalls of bugs stemming from such race conditions. The Phred system provides such determinacy information to the programmer. Various source analysis techniques have been developed for finding nondeterminacies in parallel programs. Taylor's static analysis techniques for Ada programs can be found in [24]. Techniques developed by Callahan and Subhlok [10] improve on those of Taylor by eliminating potentially exponential behavior. In [14], Emrath and Padua classify various kinds of nondeterminacy and present source analysis techniques for detecting nondeterminacies.

The issue of determinacy can also addressed from the standpoint of debugging parallel programs. Debugging parallel programs can be difficult. If race conditions exist in a parallel program, the debugging task becomes even more complicated. Multiple runs of a nondeterministic program on the same input set can produce different output sets. One solution to this problem is to trace program execution and ensure that race conditions are resolved in the same order during replay in debugging mode as during the original program execution. Methods for deterministic replay of nondeterministic programs can be found in [11, 19, 17].

1.2 Related Systems

Phred is similar to several other visual parallel programming environments, namely Code [9, 20], HeNCE [4, 5], Paralex [1, 2], and Schedule [12]. However, Phred is unique in its graph structures and its emphasis on determinacy. The intent is to develop a parallel programming environment which encourages deterministic programs without requiring programs to be deterministic.

In this paper, we describe the language (Section 2), the analysis mechanism (Section 3), and Phred programming environment.

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, using a mouse-based 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 a Phred control flow graph represents the sequential flow of control, alternation, parallelism, and synchronization.

Tokens represent the flow of control by marking nodes and edges within the control flow graph. In a sense, tokens are the instruction pointers of a Phred graph. A token residing on a task node represents a process executing the corresponding node interpretation (also called the node procedure). 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 that 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 shows the graph of a simple Phred program. 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. The 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). 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.

Phred supports complex structures for parallelism, primarily by specifying how multiple tokens may flow through the graph. Generally Phred can be used to program many parallel algorithms. Both static and dynamic algorithms have been programmed in Phred. For instance, dynamic load balancing is achieved in Phred by allowing multiple tokens to flow through a Phred graph. The tokens represent processes which obtain pieces of work from a queue and then operate on those those pieces in parallel. The tokens eventually store their results and return for more work from the work queue. A wide variety of example programs can be found in [3] and a in companion paper [6] where we give an in-depth description of numerical algorithms programmed in Phred.

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 [21], so that they can be applied to visual parallel programming in an MIMD environment. BPGs describe parallel systems and Phred programs define parallel computations. 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 through the control flow graph. 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

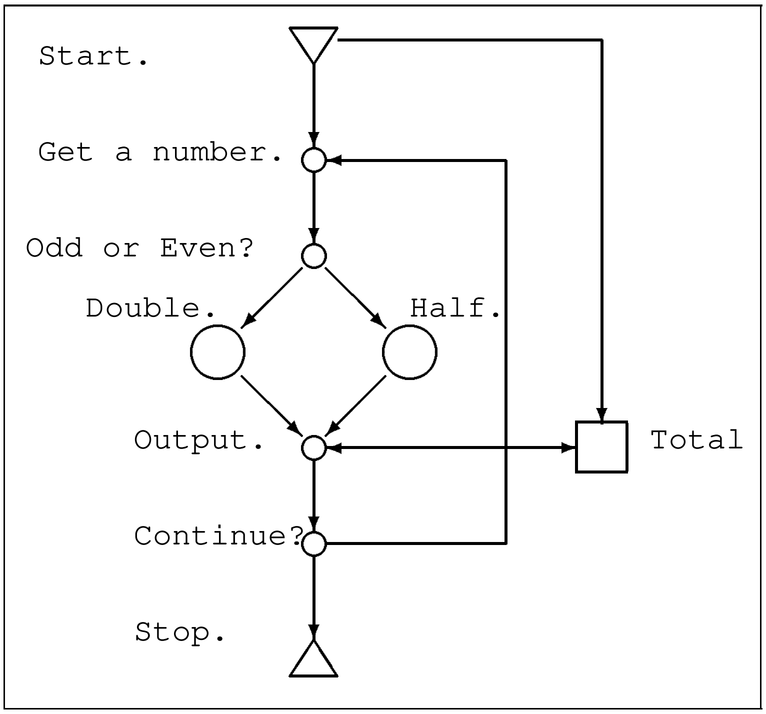

Figure 1: Sequential Phred program graph.

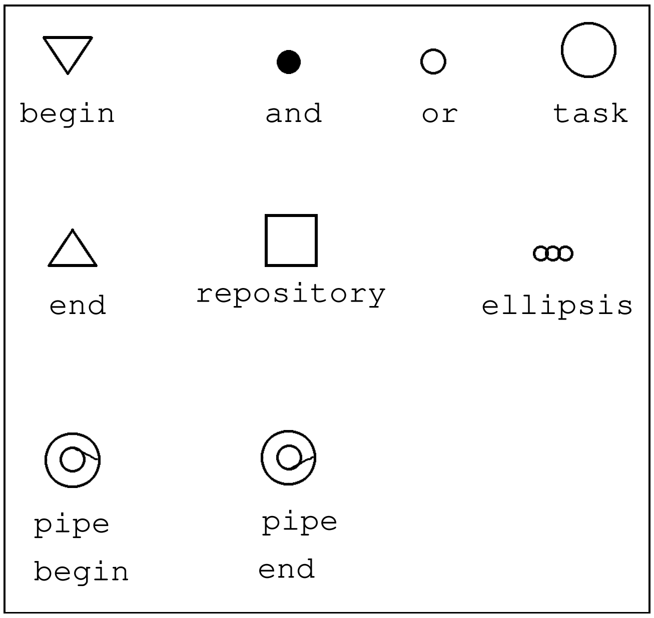

Figure 2: Legal Phred nodes.

potentially being added to the token and routed to the node's successors.

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 a node procedure is executed and written only after the procedure 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 are dynamically created and can only be accessed within the ellipsis constructs described in Section 2.3. The access patterns for data repositories are essential for analyzing the determinacy of a Phred program.

2.2 Graph Grammars

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 [13]. 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 υ is a node of α of type X, then we call υ 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.

2.3 The Phred Graph Grammar

We now use the graph grammar mechanism to describe Phred. Let **PhredCF** = (Σ, Δ, 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

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

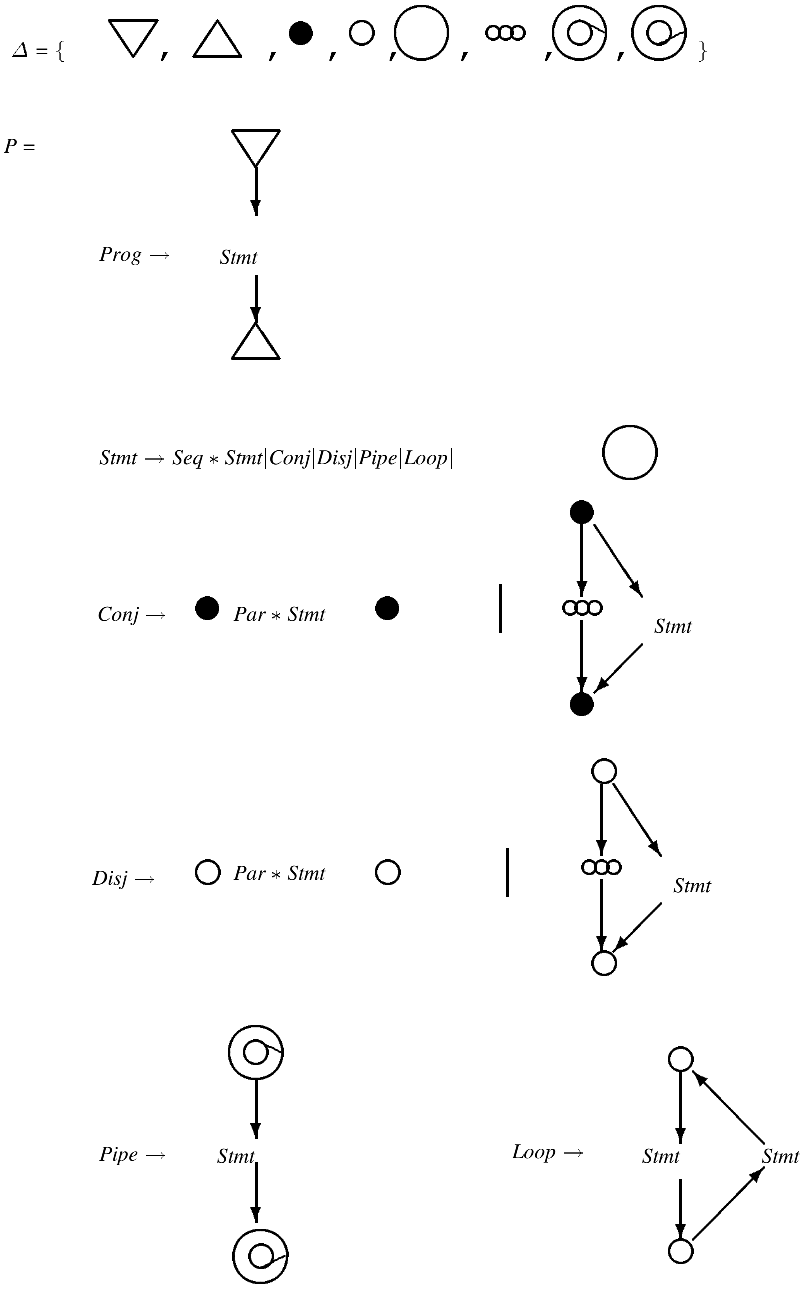

Figure 3: Phred grammar.

statement is legal. The *null* statement is a legal Phred statement. *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 (possibly different) 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 copies of the same statement to execute in parallel. The number of statements to execute is determined at run-time by calling a function from the diverging conjunctive node. The ellipsis node in the Dynamic Conjunctive construct indicates the body of the construct will be replicated multiple times. This is part of the interface (described in [3]) between the node routines and the graph.

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. As with the Dynamic Conjunctive construct, the ellipsis node indicates the body of the Dynamic Disjunctive construct will be replicated multiple times 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.

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.

As string grammars are used to describe which strings are in a particular language, graph grammars are used to describe which graphs are in a language. Now that we have described the Phred grammar we have defined the Phred language. A graph is in the Phred language if it can be derived using the the Phred grammar. Figure 4 is an example derivation of a graph in the Phred language. Various productions are applied to the nonterminal 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.

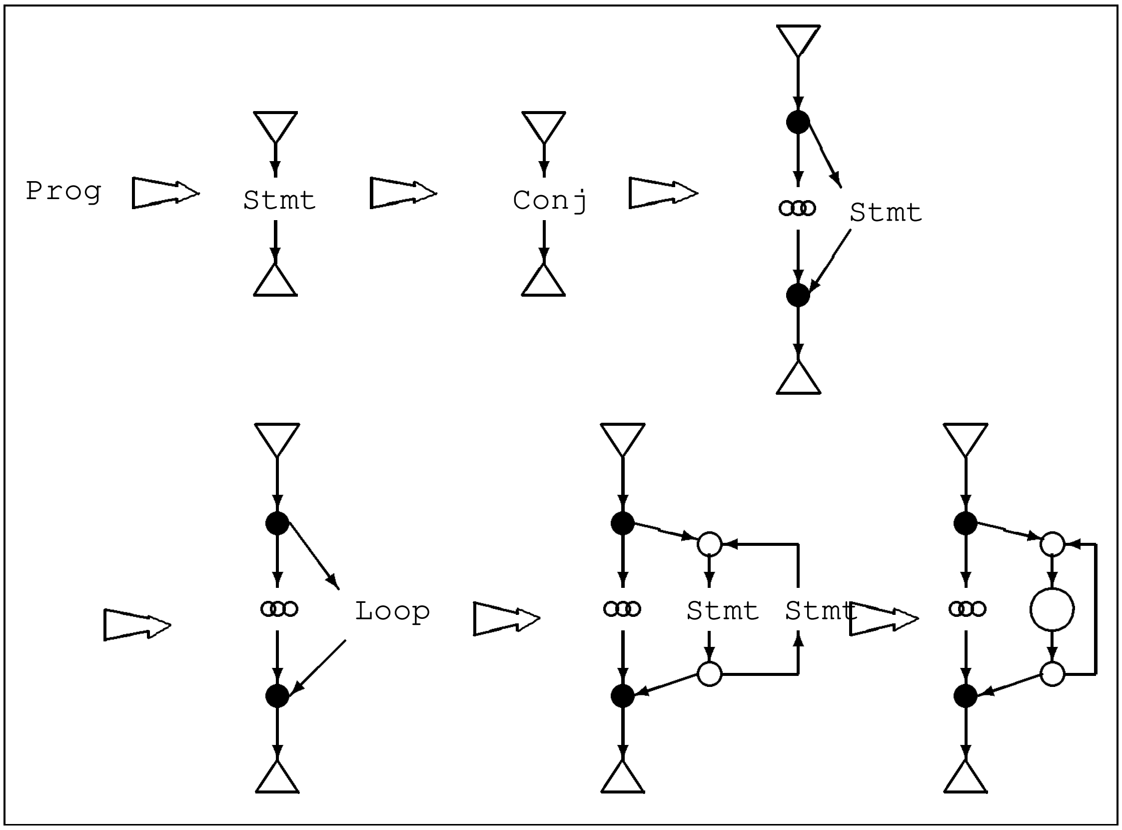

Figure 4: Example graph derivation.

2.4 An Example Parallel Program

The following example ties these concepts together. 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 matrices, using the successive overrelaxation technique. 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 in the node procedure by making a call to the `create_tokens()` routine. Thus 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 sent to the *Stop* node and the program terminates.

Thus, we have illustrated how Phred may be used to write both sequential and parallel programs, and we have given a precise definition of the Phred graphical language. The next section describes analysis theories and techniques which are used by the Phred programming tools.

3 Analysis of the Language

The Phred grammar and semantic model allow one to analyze a program for correct syntax and for determinacy. The graph parsing algorithm is a relatively straight-forward recursive descent with the exception that there is 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. Details of the parser are described in [3]. In this section, we describe the determinacy analysis of Phred graphs. This analysis is similar to the technique described by Taylor [24] for analyzing Ada programs.

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 it to produce identical output. This becomes a fundamental property for debugging programs and for using

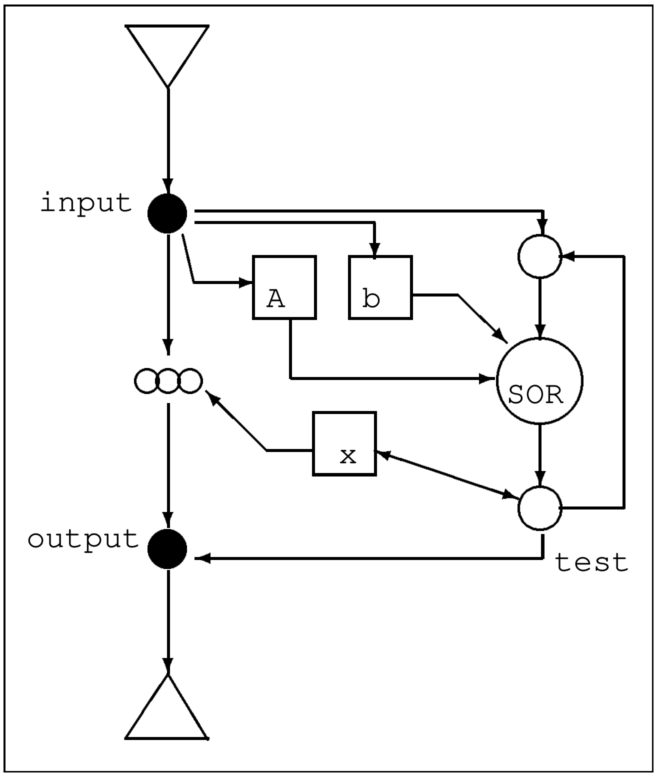

Figure 5: Parallel Phred program graph.

computers in general. 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? It may be possible to construct more efficient nondeterministic programs than deterministic ones, i.e., 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 [8] 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. These 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 [23].

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

Nondeterminacy among parallel tasks 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 accidentally 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 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 Identifying Nondeterminacy in Phred Programs

Determinacy can be viewed 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 this is not the case, then the program may or may not be determinate. If independent tasks violate Bernstein's conditions they may still be determinate. (Two tasks may write the same value to the same location, violating the conditions yet remaining determinate.) 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 that 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 that read repository r.*
- *Let W_r be the set of tasks that write repository r.*

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

$$Let\ X \leftarrow (W_r \cap p) \cup (R_r \cap p)$$

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

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

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 independent task nodes in the program that 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 that may be involved in nondeterminacy. The algorithm chooses a particular repository and a set of tasks that 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 that 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 that 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 that writes to r and at least one task in p that reads from r and these tasks are not the same ($\parallel X \parallel > 1$), 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 that 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 that 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 that 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 that 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 that 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 that may execute in parallel, can be found (Assumption 1).

3.2.1 Finding Parallel Tasks

The goal is to find all sets of nodes that could possibly run in parallel, that is, the set P from Algorithm 1. In general, tasks that 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 that 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 ith 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 that were taken to reach the node. For instance, the marking **A1B2D2** indicates several things. A node with such a marking has conjunctive nodes **A**, **B** and **D** as ancestors. Furthermore node **A** is an ancestor of node **B** and node **B** is an ancestor of node **D**. A node with such a marking can be found on the first branch from node **A** followed by the second branch from node **B** and the second branch from node **D**. Figure 6 illustrates this marking scheme. Diverging conjunctive 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.

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.

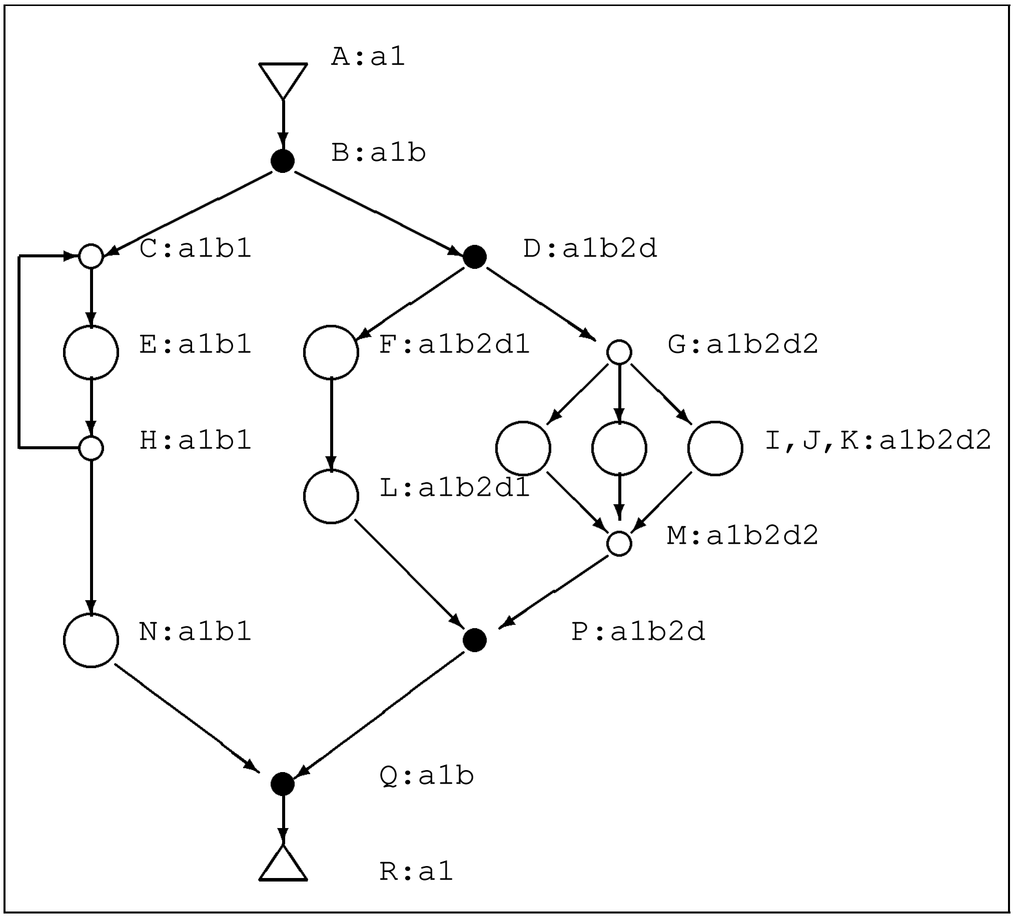

Figure 6: Example Phred graph with markings based on conjunctive node branches.

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 that 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 7 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 Pipe* and *End Pipe* nodes. If the marking of a *Begin Pipe* or *End Pipe* node have the exact same label as another node, then the nodes cannot run in parallel. For example in Figure 7, 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 Pipe* or *End Pipe* node is a prefix of the marking for another node, then these two nodes cannot execute in parallel either. In

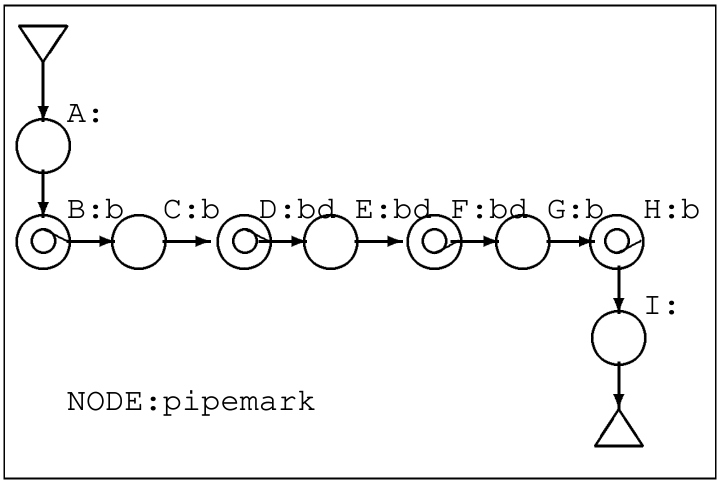

Figure 7: Example labels for pipe marking.

Figure 7, 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 3.2. 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 that 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 essentially states that use of uninitialized variables and the like 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.

The theories and algorithms developed in this section are used in the Phred programming environment described next.

4 The Phred Support System

4.1 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 window-based graphic 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 [22], specifically using several aspects of the BPG-Olympus implementation [21].

The Phred tool is itself a distributed program that provides a unique syntax-directed editor that bypasses many of the annoyances ordinarily associated with syntax-directed editors.

4.2 The Architecture

The Phred tool is designed as a collection of single *backend* and several *frontend* modules that interoperate using message-passing protocols. The backend stores the program as it is defined, and also provides a Phred execution environment. *Node interpreters* extend the backend functionality by executing procedural declarations associated with **task** nodes in the model.

A frontend performs an arbitrary filtering operation on the program stored in the backend. The most obvious example of a frontend is an interactive editor for creating and viewing the Phred graph. A frontend may also implement other functionality such as parsing and analysis of the Phred program, compilation of the program, and so on.

The frontend and the backend communicate over a general network using a specialized 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 broadcasting a similar message to all frontends indicating that a node should be added to the model. This causes the editor portion of the frontend to physically display the node. Other frontends may use this information as they see fit. Thus, 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 separate from 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.

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 simultaneously support multiple frontends (Figure 8(d)). A 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.

The components in the Phred tool are an editor-console frontend, a static analyzer frontend (called the *critic*), inspired by work of Fischer and his colleagues [15], 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 Figure 9), or at any site on the network.

The Editor-Console Frontend The Editor-Console provides a mechanism for viewing and editing Phred graphs. In general, neither the editor nor the backend check the syntax of the graph that is constructed by the user. Instead, the critic frontend accepts program descriptions as they are echoed back to the editor, and constructs and analyzes 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 intrusive,

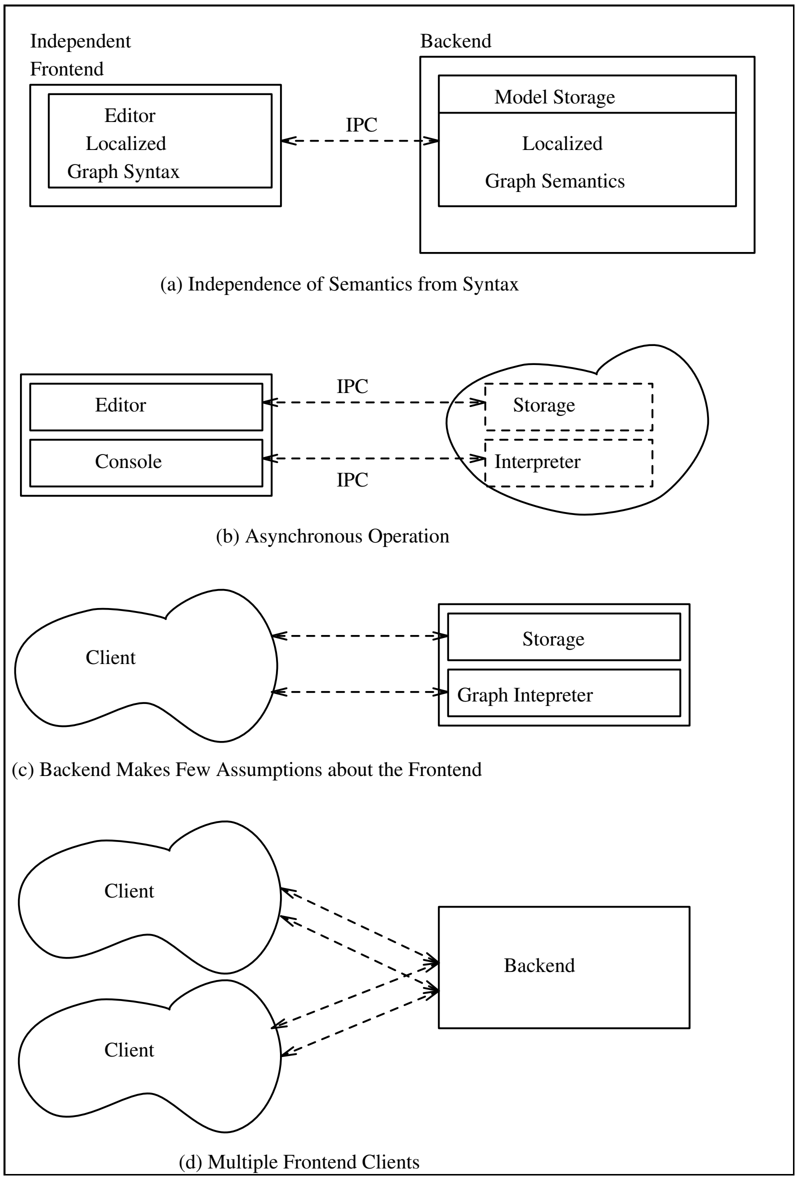

Figure 8: Views of the Olympus Architecture

since its utility will quickly become more 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, or of the 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 program.

Once the critic has successfully parsed the graph, and 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 the conjunctive control flow conditions as described in Section 3.2.

Both the parsing and analysis are nontrivial algorithms – ones that 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 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 [21]. 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 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 the C node procedures. This is accomplished by using the Sun RPC facility to bind the C procedure to the graph interpreter. We have used the low level 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 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).

4.3 Using the Tool

The tool relies on the multiwindowing environment to take advantage of the critic for syntax-directed editing. The graph server, editor, and critic are separate Unix processes that may execute on any machines accessible over a network. The editor and critic each have their own display windows.

Figure 9 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 10. The concentric circles drawn in the critic window tell the user that the graph is not syntactically correct. This version of the Phred critic does not indicate where the syntax error occurs.

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 11 is clipped image of the monochrome representation of this case; the possibly nondeterminate nodes are drawn with dashed lines.

5 Conclusions

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

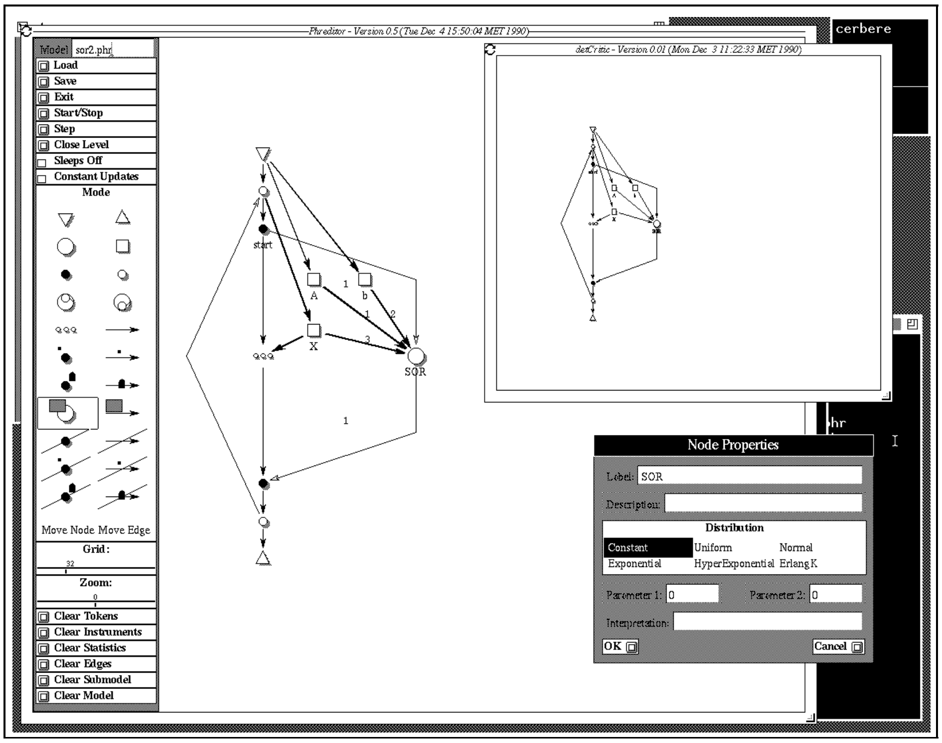

Figure 9: The Phred Editor with a Deterministic Graph

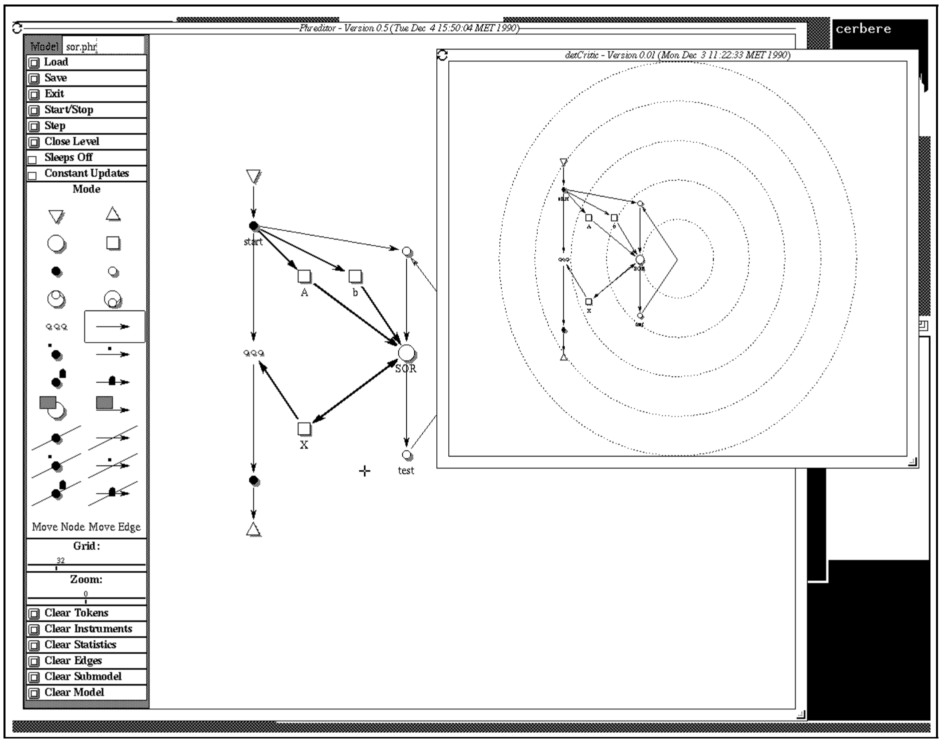

Figure 10: A Syntactically Unacceptable Graph

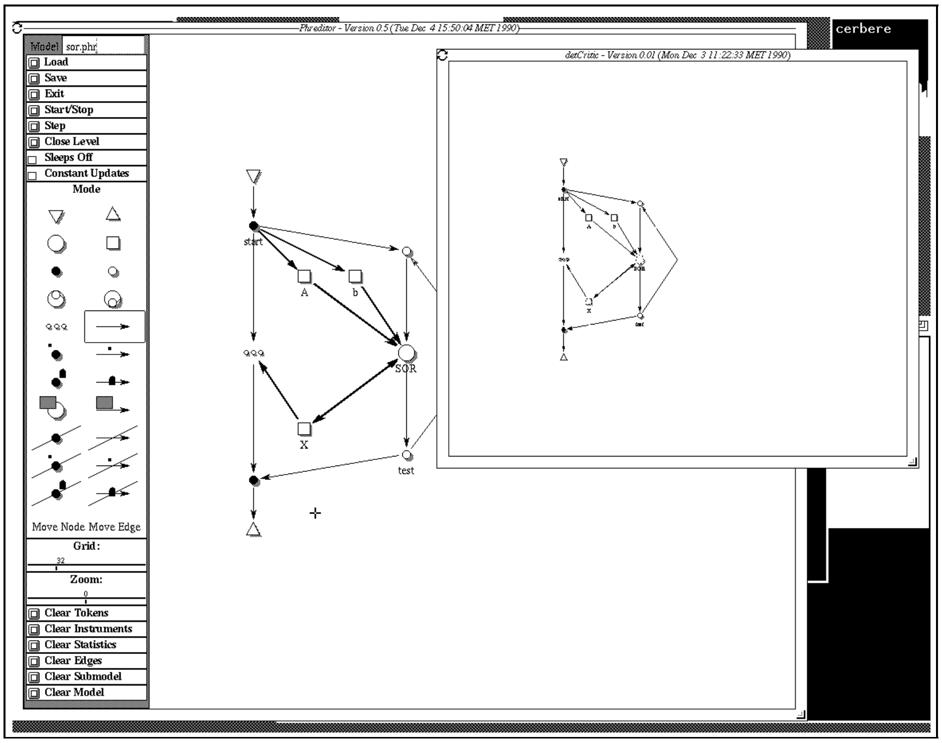

Figure 11: A Graph with Possible Nondeterminism

decision making (and synchronization), in cases where it is known that a correct result can be obtained without retaining functional behavior. Selecting a particular strategy in a game-playing algorithm is an example of this approach.

However, we believe that determinacy is generally desirable. Race conditions in nondeterministic situations are a major intellectual stumbling block for parallel programmers, much as pointers are a major stumbling block for C programmers. Just as strongly-typed languages attack the pointer problem, Phred attacks the nondeterminacy problem.

In Phred, 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) is naturally supported by the Phred graph. We expect that the software designer can use the graphs to explore algorithms qualitatively, then annotate the graphs to produce deterministic programs.

Phred has been carefully designed to incorporate components that are useful for encoding parallel algorithms, and that 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. Previous papers [3, 6] illustrate Phred's utility by presenting a variety of parallel programs expressed in Phred.

The Phred system was built as a prototype, the software was not developed to the state where it could be distributed to users. A similar follow on system, HeNCE [4, 5] is freely available. However, the HeNCE system does not yet support determinacy checking features or the data flow graph provided by Phred.

This paper describes the Phred programming language, determinacy analysis techniques, and the realization of these concepts in the Phred programming environment. Phred is unique in its combination of a visual parallel programming language, a determinacy analysis critic, and a distributed implementation of a programming environment. We hope that the concepts demonstrated by the Phred system will some day make it into production tools for parallel and distributed programming.

References

[1] O. Babaoglu, L. Alvisi, A. Amoroso, and R. Davoli. Paralex: An environment for parallel programming in distributed systems. Technical Report UB-LCS-91-01, University of Bologna, Department of Mathematics, Piazza Porta S. Donato, 5, 40127 Bologna, Italy, Febuary 1991.

[2] Ozlap Babaoglu, Lorenzo Alvisi, Alessandro Amoroso, Renzo Davoli, and Luigi Alberto Giachini. Paralex: An environment for parallel programming in distributed systems. In *1992 International Conference on Supercomputing*, pages 178–187. ACM, ACM Press, July 1992.

[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, J. Dongarra, G. A. Geist, and V. S. Sunderam. Visualization and debugging in a heterogeneous environment. *IEEE Computer*, June 1993. To appear.

[5] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of Supercomputing 91*, pages 435–444, Albuquerque, 1991.

[6] A. Beguelin and G. Nutt. Examples in Phred. In D. Sorensen, editor, *Proceedings of Fifth SIAM Conference on Parallel Processing*, Philadelphia, 1991. SIAM.

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

[8] 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.

[9] James C. Browne, Muhammad Azam, and Stephen Sobek. CODE: A unified approach to parallel programming. *IEEE Software*, 6(4):10–18, July 1989.

[10] D. Callahan and J. Subhlok. Static analysis of low-level synchronization. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, Madison,WA, May 1988. Available as Rice University, Dept. of Computer Science Technical Report TR88-70.

[11] Jong-Doek Choi and Sang Lyul Min. Race frontier: Reproducing data races in parallel-program debugging. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[12] 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.

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

[14] P. Emrath and D. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 89–99, Madison,WA, May 1988.

[15] G. Fischer, A.C. Lemke, T. Mastaglio, and A. Morch. Using critics to empower users. In *Human Factors in Computing Systems, CHI'90 Conference Proceedings (Seattle, WA)*, New York, April 1990. ACM.

[16] David Gelernter and Nicolas Carriero. Coordination languages and their significance. *CACM*, 35(2):96–107, February 1992.

[17] D. Grunwald, G. Nutt, A. Sloane, D. Wagner, and B. Zorn. A testbed for studying parallel programs and parallel execution. In *Proceedings of MASCOTS '93*, pages 95–106, San Diego, 1993.

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

[19] Robert H. B. Netzer and Barton P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of Supercomputing '92*, pages 502–511, Los Alamitos, California, November 1992. IEEE Computer Society Press.

[20] Peter Newton and James C. Browne. The CODE 2.0 graphical parallel programming language. In *1992 International Conference on Supercomputing*, pages 167–177. ACM, ACM Press, July 1992.

[21] 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.

[22] Gary J. Nutt. A simulation system architecture for graph models. In G. Rozenburg, editor, *Advances in Petri Nets*. Springer Verlag, 1990.

[23] Gary J. Nutt. *Centralized and Distributed Operating Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[24] Richard N. Taylor. A general purpose algorithm for analyzing concurrent programs. *CACM*, 26(5):362–376, May 1983.

Biographies

Adam Beguelin is an Assistant Research Professor of Computer Science at Carnegie Mellon University with a joint appointment at the Pittsburgh Supercomputing Center. In 1991 and 1992 he was a postdoctoral research associate at Oak Ridge National Laboratory and the University of Tennessee where he was also an adjunct faculty member. He received a B.S. (*summa cum laude*) in Mathematics and Computer Science from Emory University in 1985. He received his M.S. and Ph.D. degrees from the University of Colorado in 1988 and 1990 respectively. His current research interests are parallel programming environments and visualization tools for heterogeneous network computing.

Gary J. Nutt is a Professor of Computer Science and Electrical and Computer Engineering at the University of Colorado. He earned his Ph.D. in Computer Science from the University of Washington in 1972. He has also held research positions at Xerox PARC and Bell Labs, and engineering management positions at NBI and Interactive Systems. His research interests include distributed systems, operating systems, open systems, collaboration technology, performance measurement, modeling, and simulation.