FORMAL GENERATION OF EXECUTABLE ASSERTIONS FOR A
FAULT-TOLERANT PARALLEL MATRIX RELAXATION†

Hanan Lutfiyya and Bruce McMillin
CSC-91-13

October 15, 1991

Department of Computer Science
University of Missouri at Rolla
Rolla, Missouri 65401
ff@cs.umr.edu

† This work was supported in part by the National Science Foundation under Grant Numbers MIP-8909749 and CDA-8820714, and in part by the AMOCO Faculty Development Program.
Abstract

Program verification is the process of showing that a program satisfies some particular properties with respect to its specification. Axiomatic semantics is a verification method that makes assertions describing properties about the states of the program. This paper will show how executable assertions for a program can be derived from the assertions used in the verification proof of the program. While this approach has been applied to the sequential programming environment, the distributed programming environment presents special challenges. This paper focuses on applying concurrent programming axiomatic proof systems to generate executable assertions in a distributed environment using parallel matrix iterative relaxation as a model problem.

Keywords: Executable Assertions, Formal Methods, Matrix Relaxation, Concurrent Program Verification, Fault Tolerance.
I. INTRODUCTION

Multicomputers have gained much attention recently due to their unique scalability in providing massive computing power. Various hypercube multiprocessors, such as the Ncube and Intel iPSC families, are notable examples of commercially available multicomputers [AtSe88]. Design of various parallel algorithms for multicomputers has been a major research issue in the application domain. However, the design of reliable parallel algorithms has received little attention. As the size of multicomputers grows into thousands of processors and the corresponding scale of attempted application problems grows even faster, ensuring the correctness of a parallel algorithm’s computation is a necessary step in the greater use of parallel processing.

The application-oriented fault tolerance paradigm [McNi88] is a promising approach to providing this necessary fault tolerance for the multicomputer environment. In this approach, testing is performed at a peer level, i.e., processors involved in a calculation check other processors involved in different parts of the same calculation. The actual test is comprised of executable assertions [Stuc77] which are embedded in the program code to ensure that at each testable stage of a calculation, all tested processors conform to the program’s specification, design, and implementation. If a difference between this expected and observed behavior is detected, then a fault has occurred and a reliable communication of this diagnostic information is provided to the system so that appropriate actions may be taken.

Executable assertion development for a program in the multicomputer environment is complicated by several multicomputer specific issues. In a multicomputer, all processors (nodes) are autonomous with their own private local memory. The nodes are interconnected by an interconnection mechanism. There may or may not be a host processor for program downloading and data downloading/collection. Message passing is usually the only means to allow interprocessor communication, which takes non-negligible time. This limits the scope of the test that may be performed by an executable assertion to testing received messages with respect to a testing node’s local state. Thus, code or design based assertions may not be possible to implement in such an environment.

Previously, we have developed assertions for the multicomputer environment using the constraint predicate paradigm [McNi88] for applications of matrix multiplication [HoMc91], matrix iterative solution of simultaneous linear equations [McNi88], parallel relaxation labeling for the computer vision application arena [McNi88a], parallel sorting [McNi89] and the branch&bound tree search in [SuMc91]. However, this has been largely a manual task with identification of appropriate assertions based on the three basis metrics of progress, feasibility, and consistency (see Section II) of application-oriented fault tolerance. What is needed, however, to make the concept viable, is a formal method for generating executable assertions. Formal methods for the
generation of assertions is the focus of this paper. Other work focusing on the generation of assertions can be found in [LuMc91a]. In particular, we use a proof outline from program verification of a concurrent program as a starting point and generate our fault-tolerant assertions from this proof outline.

[Mili81] was the first to notice the relationship between program verification and fault tolerance through software specified executable assertions. There also exists a strong relationship between the progress and feasibility metrics of application-oriented fault tolerance and the safety and liveliness properties of concurrent program verification. However, unlike the sequential verification environment of [Mili81], complete state knowledge is not available for generating executable assertions. Thus, the additional metric of consistency, which is unique to application-oriented fault tolerance, must be an integral part of the transformation and resulting constraint predicate. We chose the global auxiliary variable axiomatic approach of [LeGr81] as a system for concurrent program verification and used the transformation outlined in [LuMc91b] from global auxiliary variables to the communication sequence-based system of [Soun84] to achieve consistency.

The remainder of the paper is organized as follows: Sections II and III briefly review the axiomatic proof systems. Section IV discusses the application-oriented fault tolerance paradigm. Section V discusses the matrix problem and iterative methods and then a verification proof. Section VI presents the main result of the paper, a transformation from the verification proof outline into a fault-tolerant program. Section VII reviews the fault-tolerant algorithm’s error coverage and Section VIII reviews the asymptotic complexity.

**II. INTRODUCTION TO PROGRAM VERIFICATION**

Mathematics provides a sound, objective way to make accurate, precise predictions about the behavior of programs. The process of using mathematics to make accurate, precise predictions about the end result of a program and then establishing that the program behaves as specified by the mathematics is called program verification. The approach examined is based on an abstraction called assertional reasoning. This approach uses the notion of a state. A program state associates a value with each variable. Execution of program results in a sequence of atomic actions, each of which transforms the state indivisibly. Assertional reasoning characterizes states by describing the properties of states. This is done by using assertions, which are formulas of predicate logic to characterize sets of states. The use of programming logic allows programs to be understood as the implementation of a relation between assertions. This is the basis of the axiomatic approach.

In the axiomatic approach, mathematical logic can be used for making assertions about the program variables before, during and after program execution. These assertions characterize
properties of program variables and relationships between them at various stages of program execution. Discussion of program verification using assertional reasoning is found in [Deut73, Floy67, HaKi57, King71, Mann74]. This paper uses the axiomatic proof system found in [LeGr81], which is used for Hoare’s model of concurrent programming, Concurrent Sequential Processes(CSP) [HoAr78].

Now to introduce the basic concepts of axiomatic semantics by defining the basic concepts.

**Definition 2.1:** An assertion is a logical statement.

**Definition 2.2:** An axiom is a true statement about the properties of a mathematical structure. A mathematical structure is defined by a set of axioms.

**Definition 2.3:** A true assertion inferred or determined from the truth of other true assertions is a theorem.

**Definition 2.4:** A proof of a theorem is an argument establishing that the theorem is true for a specified mathematical structure.

**Definition 2.5:** A rule of inference specifies the conclusions drawn from axioms and theorems known to be true. They enable the truth of certain assertions to be deduced from the truth of certain other assertions. Specifically, a rule of inference is of the following form:

$$
\frac{H_1, H_2, \ldots, H_n}{H}
$$

where \(H_1, H_2, \ldots, H_n, H\) are all assertions, has the following interpretation:

“given that \(H_1, H_2, \ldots, H_n\) are true, that \(H\) may be deduced”

Axioms, theorems and conclusions we get using a rule of inference are assertions.

A proof is often presented as a sequence of assertions such that each assertion is either an axiom of the mathematical structure, a previous theorem, or a logical inference using a rule of inference from previous steps of the proof. Therefore, to prove theorems, we must be able to make assertions about mathematical structures and to determine when one assertion follows from others.

**Definition 2.6:** For a program, the sequence of the assertions corresponding to the proof is called the proof outline.
**Definition 2.7:** The process of applying the axioms and inference rules to proving theorems is called *deductive reasoning*.

A programming language is a mathematical structure. In addition to the axioms and rules of inference of the predicate logic, the axioms and the rules of inference for this structure must characterize precisely and correctly the effect of executing programs. This is in addition to the axioms and rules of inference of the predicate logic.

The axiomatic approach to verification is based on a method that is based on assertions about the program variables before, during, and after program execution.

**Definition 2.8:** A program is *partially correct* with respect to an initial assertion I and a final assertion F if, whenever I is true of the program prior to execution of the program, and the program terminates, then F will be true of the program after the execution of the program is terminated.

**Definition 2.9:** A program is *totally correct* if it is partially correct and it can be shown that this program terminates.

Program verification requires proofs of theorems of the following type:

\[
\{P\} S \{Q\}
\]

where P and Q are assertions, and S is a statement of the language. The interpretation of the theorem is as follows: if P is true before the execution of S and if the execution of S terminates, then Q is true after the execution of S. P is said to be the *precondition* and Q the *postcondition*.

### III. A PROOF SYSTEM

The parallel programs examined are made up of component sequential processes executing in parallel. In general, to prove properties about the program, first properties of each component process are derived in isolation. These properties are combined to obtain the properties of the whole program.

The approach to showing correctness is to divide the correctness proof into two parts. The first part of the correctness proof is the sequential proofs of each individual processes that makes assumptions about the effects of the communication commands. The second part is to ensure that the assumptions are "legitimate". This will be discussed later. We will first examine the axioms and inference rules used for sequential reasoning as purposed in [LeGr81]. [LeGr81] purpouses proof rules for CSP [Hoar78]. These proof rules are similar to those found in [Hoar69].

#### 3.1 Axioms
The skip axiom is simple, since execution of the skip statement has no effect on any program or auxiliary variables.

\[
\{P\} \text{skip} \{P\} \quad \text{(skip)}
\]

The axiom states that anything about the program and logical variables that holds before executing skip also holds after it has terminated.

To understand the assignment axiom, consider a multiple assignment statement, \(\bar{x} := \bar{e}\), where \(\bar{x}\) is a list of \(x_1, x_2, \ldots, x_n\) of identifiers and \(\bar{e}\) is a list of \(e_1, e_2, \ldots, e_n\) of expressions. If execution of this statement does not terminate, then the axiom is valid for any choice of postcondition \(P\). If execution terminates, then its only effect is to change the value denoted by each target \(x_i\) to that of the value denoted by the corresponding expression \(e_i\) before execution was begun. Thus, to be able to conclude that \(P\) is true when the multiple assignment terminates, execution must begin in a state in which the assertion obtained by replacing each occurrence of \(x_i\) in \(P\) by \(e_i\) holds. This means that if \(P^{x_i}_e\) is true before the multiple assignment is executed and execution terminates, then \(P\) will be true after the assignment. Thus we have the following:

\[
\{P^{x_i}_e\} \bar{x} := \bar{e} \{P\} \quad \text{(assignment)}
\]

It may seem strange at first that the precondition should be derived from the postcondition rather than vice versa, but it turns out that this assignment rule, as well as being simple, is convenient to apply in constructing proofs about programs.

3.2 Inference Rules

There are also a rules of inference, which enable the truth of certain assertions to be deduced from the truth of certain other assertions.

A proof outline for the composition of two statements can be derived from proofs for each of its components.

\[
\frac{\{P\} S_1 \{Q\}, \{Q\} S_2 \{R\}}{\{P\} S_1 ; S_2 \{R\}} \quad \text{(composition)}
\]

When executing \(S_1; S_2\), if \(Q\) is true when \(S_1\) terminates it will hold when \(S_2\) starts. From the second hypothesis, if \(Q\) is true just before \(S_2\) executes and \(S_2\) terminates, then \(R\) will hold. Thus if \(S_1\) and \(S_2\) are executed one after the other and \(P\) holds before the execution, then \(R\) holds after the execution.

† This stands for predicate \(P\) with each \(x_i\) replaced with \(e_i\)
Execution of an alternate command ensures that a statement $S_i$ is executed only if its guard $b_i$ is true. Thus, if an assertion $P$ is true before execution of the alternate command, then $P \land b_i$ will hold just before $S_i$ is executed. The second part of the hypothesis assumes that none of the guards are true. If the hypothesis is true and if the alternate statement terminates, then this is sufficient to prove that $Q$ will hold should the alternate statement terminate.

$$\forall i: \{P \land b_i\} c_i; S_i \{Q\}, \{P \land \forall i: \neg b_i\} \rightarrow \{Q\}$$

(alteration)

The consequence rule allows the precondition of a program or part of a program to be strengthened and the postcondition to be weakened, based on deduction possible in the predicate logic.

$$P \rightarrow P', \{P'\} S \{Q'\}, Q' \rightarrow Q$$

(consequence)

Parallel programs are composed of a set of communicating sequential processes. In many programs, it is desirable to save part of the communication sequence between processes. This is done with the use of "dummy" or auxiliary variables. The need for such variables has been independently recognized by many. The first reference that shows the usefulness of auxiliary variables is found in [Clin73]. The auxiliary variables must not affect program control during execution. The following rule allows us to draw conclusions from proof outlines of programs annotated with auxiliary variables.

$$\{P\} S \{Q\}$$

(auxiliary variables)

where $S$ is obtained from $S'$ be deleting all references to auxiliary variables and $P$ and $Q$ do not contain any free variables that are auxiliary variables.

The inference rule for the repetition command is based on a loop invariant i.e. an assertion that holds both before and after every iteration of a loop.

$$\forall i: \{P \land b_i\} c_i; S_i \{P\}$$

(repetition)

The hypothesis of the repetition rule requires that if execution of $S_i$ is begun when the assertion $P$ and $b_i$ is true, and if execution terminates, then $P$ will again be true. Hence, if an assertion $P$ is true just before the execution a repetition command, then $P$ is true at the beginning and end of each iteration. Thus, $P$ will hold if the repetition terminates. The repetition ends when no boolean guard is true, so $\neg b_1 \land \neg b_2 \land \cdots \neg b_n$ will also hold at that time.
[LeGr81] does not have distributed termination which is contradictory to Hoare’s original version of CSP [Hoar78]. Distributed termination provides the means for automatic termination of a loop in one process because another process has terminated. It is assumed that all termination of loops occurs when all boolean guards are false.

3.3 Axioms and Inference Rules Dealing With Communication

The communication axiom is as follows:

\[
\{P\} \alpha \{Q\} \quad \text{(communication)}
\]

where \( \alpha \) is a communication command.

Remember that \( \{P\}S\{Q\} \) means total correctness if \( S \) terminates. \( S \) terminates in the absence of deadlock. The parallel rule implies that a proof for it is based on the isolated sequential proofs of the programs it comprises. Take any such program \( S \). A sequential proof for it only proves facts about it running in isolation. With only one process running, communication commands deadlock. Thus, any predicate \( Q \) may be assumed to be true upon termination of a communication command because termination never occurs.

The Law of the Excluded Miracle [Dijk76] states that the statement \( \{\text{false}\} \) should never be derived. This is the requirement to ensure a sound statement. The communication axiom does violate the Law of the Excluded Miracle. This allows us to deduce that the following is true:

\[
\{\text{true}\} \text{A?x}\{x = 5 \land x = 6\}
\]

The postcondition, however, is obviously false. Thus, one might come to the conclusion that the proof system is not sound. This is the result of allowing the communication axiom to make assumptions about the behavior of other processes to prove properties of an individual process. To justify those assumptions a "satisfaction proof" must be done. This ensures that the proof system is sound. Hence, there parallel interference rule is as follows:

The parallel inference rule is the following:

\[
\begin{align*}
\forall i: \{P_i\} S_i\{Q_i\} \text{ satisfied and inference – free} \\
\{ \forall i: P_i \} \parallel_{i=1:n} \{ \forall i: Q_i \}\}
\end{align*}
\]

The parallel rule implies that construction of the proof of a parallel program can be derived from the partial correctness properties of the sequential programs it comprises.

It has been mentioned that a "satisfaction proof" is needed to ensure soundness of the proof system. Let us examine the proof outline of the matching communication pair.

\[
\begin{align*}
P1: \text{[ \cdots \{P\}P2\?x\{Q\}] } \\
P2: \text{[ \cdots \{R\}P1!y\{S\}]}
\end{align*}
\]
The effect of these two communication commands is to assign \( y \) to \( x \). This implies that \( Q \land S \) is true after communication if and only if

\[
\text{implies that } Q \land S \text{ is true after communication if and only if}
\]

A "satisfaction proof" is such that the above is proven for every matching communication pair. This is called the rule of satisfaction.

Earlier we discussed the need for auxiliary variables. An auxiliary variable may affect neither the flow of control nor the value of any non-auxiliary variables. Otherwise, this unrestricted use of auxiliary variables would destroy the soundness of the proof system. Hence, auxiliary variables are not necessary to the computation, but they are necessary for verification. [LeGr81]'s proof system allows for auxiliary variables to be global i.e. variables that can be shared between distinct processes. Auxiliary variables are used to record part of the history of the communication sequence. Shared reference to auxiliary variables allow for assertions relating the different communication sequences. This necessitates the need for a Proof of Non-interference. This consists of showing that for each assertion \( T \) in process \( P_i \), it must be shown that \( T \) is invariant over any parallel execution. This is the non-interference property of [Owic75].

Other axiomatic proof systems are included in [ApRo81] and [Soun84]. The proof system in [ApRo81] uses local auxiliary variables. The proof system in [Soun84] uses communication sequences. A communication sequence for process \( i \) is the sequence of all communications that process \( i \) has so far participated in. Each process \( i \) has a variable denoting the its communication sequence, which is updated for each communication. This allows for proof rules that can make inferences about the communications sequences. Thus, it is sufficient to do only sequential proofs of each component process in a parallel program. Our work in [LuMc91b] shows that these three proof systems are equivalent in that they allow for the same properties to be proven. However, it is easier to reason in some systems more than others.

### IV. BACKGROUND ON APPLICATION-ORIENTED FAULT TOLERANCE

Application-oriented techniques rely on the creating and embedding of executable assertions into program code. Development of these assertions is straightforward for the sequential program execution environment (See [YaCh75, Andr79, Somm82, HuAb87] for examples). These works concentrate on generating from the coding and design process.

Assertions integrated into the program code are called executable assertions. Assertions are embedded as follows:
if not <assertion> then ERROR

If an assertion fails as the result of an abnormality, then the ERROR condition is raised and the appropriate action is taken.

Earlier work in selecting executable assertions for the application-oriented fault tolerance paradigm was guided by “Natural Constraints” of the problem[McNi88]. Working from these natural constraints starting at the specification phase provides a global perspective on assertion development. The parallel decomposition of the problem ties in closely with the design of the program to solve the problem. Thus, the executable assertions are closely tied to the individual components of the life cycle.

A set of high-level basis metrics were used to extract assertion generating properties from the problem specification. In a solution, each testable intermediate result should satisfy one or more of the subclasses of progress, feasibility, and consistency. The three subclasses form the class of constraint predicates in which each subclass contributes to form a cohesive unit that provides error coverage. The constraint predicate satisfies both the “liveliness” and “safety” properties of [Hail82] and [Lamp77]. Liveliness ensures that the solution makes progress - i.e. does not stall or deadlock. Safety ensures that the solution obtained will be the correct solution.

As will be shown in this paper, each of these subclasses of executable assertions can be derived from the verification proof. The progress and feasibility components of the constraint predicate can be directly derived from a subset of assertions used in the verification proof. Progress corresponds to those assertions describing liveliness properties. Feasibility corresponds to those assertions describing safety properties. Testing is done through executable assertions where the the state of the program must satisfy the subset of assertions chosen for progress and feasibility.

It is necessary to ensure that any two processors that are correct, given the same input values, executing the same executable assertion, reach the same conclusion of error or no error. However, in a system that has Byzantine faults, it is possible for a faulty processor to send inconsistent messages to different processors. It is necessary for this to be detected. This is the purpose of the consistency condition. Mathematically, this can be described as follows:

**Definition 4.1:** For processes i and j, a communication path P from j to i is a sequence of processes labelled j, \(l_1, \ldots, l_n, i\), where \(l_1, \ldots, l_n\) denotes the path of processes between j and i through which a value computed in process j is communicated to process i.
**Definition 4.2:** For a process $i$, let $h_i$ denote the sequence of all communications that process $i$ has so far participated in as the receiving process. Then $h_i$ is a list consisting of tuples of the following form:

$$<j, (\text{Var}, \text{Val}), T, P>$$

where

- $j$ is a process that $i$ receives from.
- $\text{Var}$ is the variable that $j$ is transmitting to $i$.
- $\text{Val}$ is the value of $\text{Var}$.
- $T$ denotes the time at which variable $\text{Var}$ had the value denoted by $\text{Val}$.
- $P$ denotes the communication path from $j$ to $i$.

In general, consistency can then be defined by the following conditions:

**Definition 4.3:** For a non-faulty process $i$, if exists any two tuples $t_1, t_2 \in h_i$ such that

$$t_1 = <j, (\text{Var}, \text{Val}_1), T, P_1>$$
$$t_2 = <j, (\text{Var}, \text{Val}_2), T, P_2>$$

then if $\text{Val}_1 \neq \text{Val}_2$ the system is said to be **inconsistent** otherwise the system is said to be **consistent**.

It can be shown that if the value of a variable computed in time $T$ is communicated to a set of processes on more than one path, then under a bounded number of faults the consistency definition of 4.2 ensures that the non-faulty processes in the set receive the same values of variables. This, in turn, ensures that for any two processors that are correct, executing the same executable assertion, reach the same conclusion. This will be shown in Section VI.

This definition of consistency is general and indeed the strict notion of time $T$ can be relaxed significantly in applications. This definition is meant as a guideline towards defining consistency for specific applications.

This paper provides a method of translating a verification proof for a parallel matrix relaxation algorithm into a fault tolerant algorithm.

**V. VERIFICATION PROOF OF A MATRIX ITERATIVE ALGORITHM**
Matrix iterative techniques have been employed since the 1940’s to solve large systems of equations numerically [Varq62, Ames77]. These systems of equations arise out of the numerical solution of physical problems utilizing finite difference and/or finite element methods. Techniques of particular interest are the pointwise relaxation techniques. While not as elegant as some other methods, they do contain the massive inherent parallelism necessary to take full advantage of a multicomputer.

The problem is to solve the linear system \( Au = v \), where, \( A = (a_{i,j}) \) is a nonsingular \( Q \times Q \) complex matrix, \( v = (v_i) \) is a complex vector, and \( u = (u_i) \) is the solution vector for \( i, j \in \{1, 2, \ldots, Q\} \) and \( Q \) a perfect square. The method of Gauss-Seidel Relaxation (GSR) is an iterative technique used to obtain an approximate solution, \( u^{(K)} = (u_i^{(K)}) \), where \( K \) is the final iteration, to this system. To parallelize an inherently sequential solution technique, a consistent multicolor ordering of the \( Q \) values of \( u^{(K)} \) (also called points) must be made. Assume \( Q = N \) for simplicity and let process \( i \), \( i = 1, \ldots, N \) compute value \( u_i \). Selection of points for computation is controlled by the iteration variable \( k \) as in the following predicate.

**Definition 5.1:**

\[
\Lambda_{i,k} = \begin{cases} 
1 & \text{if } P_i \text{ is active during iteration } k \\
0 & \text{otherwise.}
\end{cases}
\]

The most common of these consistent multicolor orderings is the red/black or checkerboard multicolor ordering [Smit65] (implying, among other things, that the number of nonzero \( a_{i,j} \)'s per row \( i \) is no more than five). For this specific case \( \Lambda_{i,k} \) becomes:

**Definition 5.2:**

\[
\Lambda_{i,k} = \begin{cases} 
1 & \text{if } (i \ \text{div} \ \sqrt{N} + i \ \text{mod} \ \sqrt{N}) \text{ is even and } k \text{ is even for } i \in \{1, \ldots, N\}, k \in \{0, 1, 2, \ldots\} \\
1 & \text{if } (i \ \text{div} \ \sqrt{N} + i \ \text{mod} \ \sqrt{N}) \text{ is odd and } k \text{ is odd for } i \in \{1, \ldots, N\}, k \in \{0, 1, 2, \ldots\} \\
0 & \text{otherwise}
\end{cases}
\]

where \( \text{div} \) is the integer division operator and \( \text{mod} \) is the remainder operator for integer division. The \( u_i \)'s are assigned to points of a \( \sqrt{Q} \times \sqrt{Q} \) coordinate indexed grid such that \( u_i \)'s with nonzero \( a_{i,j} \)'s are within distance one of \( u_i \). Each \( u_i \) is updated on odd/even values of \( k \) according to \( \Lambda_{i,k} \). An odd/even iteration cycle is sometimes called two half-iterations. This paper considers (without loss of generality) consistent multicolor orderings of length two. The pointwise GSR method is given by the following equation:
\[ u_i^{(k)} = -\frac{1}{a_{i,i}} \left[ \sum_{j \neq i} a_{i,j} u_j^{(k-1)} - v_i \right] \quad \text{if} \quad \Lambda_{i,k} = 1 \quad \text{(5.1)} \]

In the case of a two-dimensional grid, we can think of the values of \( u^{(K)} \) on the right hand side of (5.1) as neighbors of the point \( u_i \) at the compass directions North, South, East, and West. Each new value of a point is iteratively a function of the current values of these four neighbors, the locations of which are North: \((i \div \sqrt{Q} + 1, i \mod \sqrt{Q})\), South: \((i \div \sqrt{Q} - 1, i \mod \sqrt{Q})\), East: \((i \div \sqrt{Q}, i \mod \sqrt{Q} + 1)\), and West: \((i \div \sqrt{Q}, i \mod \sqrt{Q} - 1)\).

**Definition 5.3:** For a point \( u_{\text{node}} \), where \( 1 \leq \text{node} \leq Q \), \( \text{node} \) denotes the coordinate location of \( u_{\text{node}} \) in the two dimension grid. \( \text{node} \) is equal to \((\text{node} \div \sqrt{Q}, \text{node} \mod \sqrt{Q})\).

**Definition 5.4:** For a process labelled \( \text{node} \), if \( I \in \{ \text{North}, \text{South}, \text{West}, \text{East} \} \), then \( I_{\text{North}}, I_{\text{South}}, I_{\text{West}}, \) and \( I_{\text{East}} \) are the neighbors of \( I \) at the compass directions North, South, West, East.

The program fragment of Figure 5.1 shows typical relaxation calculation. Each processor runs the same algorithm.
Procedure Relax($u_i^{(0)}$)
/* For Processor at location $i$ */
/* $u_i$ is the data held by $i$ */

$k \leftarrow 0$  /* Iteration Count */
/* $u_i^0$ is the initial value */

while $|u_i^{(k+2)} - u_i^{(k)}| \leq \epsilon$
    foreach (direction in {North, South, East, West})
        if ($\Lambda_{i,k} = 1$)
            receive $u_i^{(k)}$ from $P_{direction}$;
        else
            send $u_i^{(k)}$ to $P_{direction}$;
        if ($\Lambda_{i,k} = 1$)
            $u_i^{(k+1)} \leftarrow \frac{1}{a_{i,i}} \left[ \sum_{j \neq i} a_{i,j} u_j^{(k)} - v_i \right];$
        $k \leftarrow k + 1$;

Figure 5.1: Relaxation Skeleton
D is the set {North, South, West, East}.

Now to discuss the verification of the above program. First, we will discuss the postcondition, which is an assertion that must be true on the while loop exit. The postcondition (labelled Post) is the following:

Post: $Au^{(K)} = v - \epsilon \land$

$\forall i: |u_i^{(K)} - u_i^{(K-2)}| < \epsilon$

where $K$ represents the last iteration. In other words, we want the computed solution to converge within $\epsilon$ of the true solution.

To formulate the precondition, we present the following theorem to characterize matrices which can be solved by (5.1).

**Theorem 5.1:** [Varg62] Let $A$ be a strictly or irreducibly diagonally dominant Hermitian $Q \times Q$ complex matrix. Then the pointwise GSR method is convergent for any initial assignment vector $u_i^{(0)}$ under a consistent multicolor ordering.
Theorem 5.1 shows the conditions matrix A must satisfy to ensure that there is a solution. This ensures that a solution does exist. This will become part of the precondition of the program in this paper.

For our purposes, it is useful to derive the conditions under which Equation (5.1) monotonically converges. These conditions will be included as part of the precondition for the relaxation algorithm.

**Theorem 5.2:** Let A be a matrix satisfying Theorem 5.1 with positive diagonal and nonpositive off-diagonal entries. Then the solution given by (5.1) will converge monotonically to the final solution $u^{(k)}$ for the following choices of $u^{(0)} = (u_i^{(0)})$.

1) Let $u_{\text{min}}$ be the smallest value such that $v_i - u_{\text{min}} \sum_{j=1}^{Q} a_{ij} \leq 0$ for all $i \in \{1, 2, \ldots, Q\}$.

   If $u_i^{(0)} = u_{\text{min}}$ for all $i \in \{1, 2, \ldots, Q\}$, then $u_i^{(k)} \geq u_i^{(k+2)}$ for all $k \in \{0, 1, \ldots\}$.

2) Let $u_{\text{max}}$ be the largest value such that $v_i - u_{\text{max}} \sum_{j=1}^{Q} a_{ij} \geq 0$ for all $i \in \{1, 2, \ldots, Q\}$.

   If $u_i^{(0)} = u_{\text{max}}$ for all $i \in \{1, 2, \ldots, Q\}$, then $u_i^{(k+2)} \geq u_i^{(k)}$ for all $k \in \{0, 1, \ldots\}$.

**Proof:** For case 1), by induction on $k$, the iteration count.

Basis, $k=0$,

$$u_i^{(1)} = \frac{1}{a_{ii}} \left[ v_i - \sum_{j \neq i}^{Q} a_{ij} u_i^{(0)} \right]$$  \hspace{1cm} (5.2)

and since

$$v_i - \sum_{j=1}^{Q} a_{ij} u_i^{(0)} \leq 0$$

then

$$v_i - \sum_{j \neq i}^{Q} a_{ij} u_i^{(0)} - a_{ii} u_i^{(0)} \leq 0$$  \hspace{1cm} (5.3)
Since $a_{i,i} > 0$, dividing (5.3) by $a_{i,i}$, simplifying and substituting in (5.2) yields

$$u^{(1)}_i = \frac{1}{a_{i,i}} \left[ v_i - \sum_{j \neq i}^Q a_{i,j} u^{(0)}_j \right] \leq u^{(0)}_i$$

(5.4)

Continuing to the completion of the half iteration pair,

$$u^{(2)}_i = \frac{1}{a_{i,i}} \left[ v_i - \sum_{j \neq i}^Q a_{i,j} u^{(1)}_j \right]$$

By (5.4) and since $a_{i,j} \leq 0, i \neq j; i, j \in \{1, 2, \ldots, Q\}$

$$\leq \frac{1}{a_{i,i}} \left[ v_i - \sum_{j \neq i}^Q a_{i,j} u^{(0)}_j \right]$$

By (5.3)

$$\leq u^{(0)}_i$$

Now assume $u^{(l)}_i \leq u^{(l-2)}_i$ for $l = 2, 3, \ldots, k - 1, k$. Then for $l = k + 1$

$$u^{(k+1)}_i = \frac{1}{a_{i,i}} \left[ v_i - \sum_{j \neq i}^Q a_{i,j} u^{(k)}_j \right]$$

Since $a_{i,j} \leq 0, i \neq j$,

$$\leq \frac{1}{a_{i,i}} \left[ v_i - \sum_{j \neq i}^Q a_{i,j} u^{(k-2)}_j \right]$$

$$\leq u^{(k-1)}_i$$

The proof for case (2) is symmetric. □

To simplify the verification proof, we will assume for the precondition that case 1 of Theorem 5.2 is true.

As part of a safety property, only iteration values that are feasible solutions of the system are allowed.
**Theorem 5.3:** In the solution of (5.1), each \( u^{(k)}_i, i \in \{1, 2, \ldots, Q\}, k \in \{0, 1, \ldots\} \) is bounded by \( u_{\min} \leq u^{(k)}_i \leq u_{\max} \) for \( u^{(k)}_i \), \( u^{(k+2)}_i \) satisfying the conditions of Theorem 5.2.

**Proof:** Assume that for some \( u^{(k)}_i, u_{\min} < u^{(k)}_i \) or \( u^{(k)}_i < u_{\max} \). Then there are two conditions based on the initial assignment vector \( u^{(0)} \).

1) Let the initial assignment be \( u^{(0)}_i = u_{\min}, i \in \{1, \ldots, Q\} \). One of two cases can occur.

By Theorem 5.2 \( u^{(k)}_i \leq u^{(k-2)}_i, i \in \{1, \ldots, Q\}, k \in \{2, 3, 4, \ldots\} \). Thus

\[
\begin{align*}
  u^{(k)}_i &\leq u^{(k-2)}_i \leq \cdots \leq u^{(0)}_i \quad \text{for } k \text{ even} \\
  u^{(k)}_i &\leq u^{(k-2)}_i \leq \cdots \leq u^{(1)}_i \quad \text{for } k \text{ odd}
\end{align*}
\]

\( \leq u^{(0)}_i \) by (5.4)

Alternately assume that \( u^{(k)}_i < u_{\max} \) for some \( i \in \{1, 2, \ldots, Q\} \) and \( k \in \{0, 1, 2, \ldots\} \) and \( u^{(l)}_i \geq u_{\max} \) for \( l = 0, \ldots, k - 1 \). By (5.1)

\[
  u^{(k)}_i = \frac{1}{a_{i,i}} \left[ v_i - \sum_{j \neq i} a_{i,j} u^{(k-1)}_j \right] < u_{\max}
\]

or

\[
  v_i - \sum_{j \neq i} a_{i,j} u^{(k-1)}_j - a_{i,i} u_{\max} < 0
\]

The choice of \( u_{\max} \) satisfies

\[
  v_i - \sum_{j \neq i} a_{i,j} u_{\max} - u_{\max} \geq 0
\]

Rearranging each yields

\[
\begin{align*}
  \sum_{j \neq i} a_{i,j} u^{(k-1)}_j &> v_i - a_{i,i} u_{\max} \\
  \sum_{j \neq i} a_{i,j} u_{\max} &\leq v_i - a_{i,i} u_{\max}
\end{align*}
\]
Substituting

\[ \sum_{j \neq i}^{Q} a_{ij}u_{j}^{(k-1)} > \sum_{j \neq i}^{Q} a_{ij}u_{\text{max}} \]

Since \( a_{ij} \leq 0 \) for \( i, j \in \{1, 2, \ldots, Q\} \), \( u_{j}^{(k-1)} < u_{\text{max}} \) for at least one \( u_{j}^{(k-1)} \), a contradiction.

2) The case for the initial choice of \( u_{\text{max}} \) is symmetric. \( \square \)

The results of Theorem 5.2 and Theorem 5.3 will become part of the loop invariant for the loop of the program presented in Figure 5.1. Also included in the loop invariant is that the residuals of the points computed in each iteration is zero. Figure 5.2 is a proof outline for the program presented in Figure 5.1.
**Procedure Relax**($u_i^{(0)}$)
/* For Processor at location i */
/* $u_i$ is the data held by i */

$k \leftarrow 0$ /* Iteration Count */
/* $u_i^0$ is the initial value */

{Loop $i$}

while $|u_i^{(k+2)} - u_i^{(k)}| \leq \varepsilon$

{Loop $i$}

foreach (direction in {North, South, East, West})

if ($A_{i,k} = 1$)

    receive $u_{i}^{(k)}$ from $P_{direction}$;

else

    send $u_{i}^{(k)}$ to $P_{direction}$;

<∀direction in $D$ $pu_{direction} \geq cu_{direction} \land u_{direction} = cu_{direction}>

if ($A_{i,k} = 1$)

    $u_i^{(k+1)} \leftarrow -\frac{1}{a_{i,i}} \left[ \sum_{j \neq i} a_{i,j} u_j^{(k)} - v_i \right]$;

{ $u_i = -\frac{1}{a_{i,i}} \left[ \sum_{j \neq i} a_{i,j} cu_j - v_i \right] \land \forall direction in D pu_{direction} \geq cu_{direction} \land u_{direction} = cu_{direction}$ }

$pu_i = cu_i$;
$cu_i = u_i$;

$k \leftarrow k + 1$;

{Post}

---

**Figure 5.2:** Proof Outline

$D$ is the set {North, South, West, East}.

---

The following auxiliary variables are being used:

$cu_i$ This denotes the current value of $u_i$.

$pu_i$ This denotes the previous value of $u_i$.

$u_{\text{min}}$, $u_{\text{max}}$

Derived from Theorem 5.2.

The precondition (labelled Pre), which was derived from the conditions of Theorems 5.1, 5.2 and 5.3, is the following:
Pre: \( \exists u_{\text{ue} R^n} \ A u^* = f \) \( \wedge \)

\( u_{\text{min}} \) is the smallest value such that \( v_i - u_{\text{min}} \sum_{j=1}^{Q} a_{i,j} \leq 0 \ \forall \ i \in \{1, 2, \ldots, Q\} \) \( \wedge \)

\( u_{\text{max}} \) is the largest value such that \( v_i - u_{\text{max}} \sum_{j=1}^{Q} a_{i,j} \leq 0 \ \forall \ i \in \{1, 2, \ldots, Q\} \) \( \wedge \)

\( u_i^{(0)} = u_{\text{min}} \ \forall \ i \in \{1, 2, \ldots, Q\} \)

**Definition 5.5:** Let \( r_i^{(k)} = v_i - \sum a_{ij} u_j^{(k)} \) denote the residuals at step \( k \).

The loop invariant is the following (labelled Loop_\( i \)):

\[
\begin{align*}
  k > 2 & \rightarrow (p u_i \geq c u_i \land u_{\text{min}} \geq c u_i \geq u_{\text{max}}) \land \\
  A_{i,k-1} = 1 & \rightarrow (u_i = c u_i \land r_i^{(k)} = 0) \land \exists u_{\text{ue} R^n} \ A u^* = f
\end{align*}
\]

It should be noted that the loop invariant must be true before the first execution of the loop and after incrementing the value of \( k \) inside the loop.

Let \( c u_i \) denote the value of \( u_i^{(k+2)} \) and \( p u_i \) denote the value of \( u_i^{(k)} \). If we assume case 1 from Theorem 5.2, then if (5.1) is to converge monotonically, then for each \( k \) and for each \( i \in \{1, \ldots, Q\} \), then either \( u_i^{(k)} \geq u_i^{(k+2)} \) or in other words \( p u_i \geq c u_i \). The condition \( u_{\text{min}} \geq c u_i \geq u_{\text{max}} \) is similarly derived from Theorem 5.3.

The postcondition (labelled Post) is the following:

Post: \( A u^{(K)} = v - \epsilon \) \( \wedge \)

\[
\forall i: |u_i^{(K)} - u_i^{(K-2)}| < \epsilon
\]

where \( K \) represents the last iteration.

Convergence and the loop invariant imply the postcondition. The verification proof showed that the solution is monotonically converging i.e. for each \( k \) we have that the following is true:

\[
u_i^{(k+2)} \leq u_i^{(k)}
\]

This implies that

\[
\sqrt{\sum (u_i^{(k+2)} - u_i^*)^2} \leq \sqrt{\sum (u_i^{(k)} - u_i^*)^2}
\]

Or in other words,
\[ \|e_{k+2}\| \leq \|e_k\| \]

where \(\|e_{k+2}\|, \|e_k\|\) denote the error norm in stages \(k+2\) and stage \(k\) respectively. Since, the GSR method can be shown to be iterative convergent for the preconditions defined in this section, we can assert the stronger condition that \(\|e_{k+2}\| < \|e_k\|\). Therefore, it easy to see that if the solution is monotonically decreasing then the error norm decreases at each step. The discussion in [Ames77a] shows that if there exists a solution, and the error norm decreases in each step then it can cease decreasing only when \(u_i^{(K)} = u_i^{(K-2)}\) for all \(i\). When this occurs \(u^{(K)} = u^*\). However, this program will stop decreasing when for all \(i\)

\[ |u_i^{(K)} - u_i^{(K-2)}| \leq \varepsilon \]

There are two cases. Either the residual \(r_i\) is 0 or \(r_i \leq \varepsilon\). This implies that \(Au^{(K)} = v - \varepsilon\).

VI. TRANSLATING THE PROOF OUTLINE OF THE RELAXATION ALGORITHM TO FAULT TOLERANT CONSTRAINTS

For the sake of simplicity, the analysis is for the interior nodes. The border nodes are special cases.

A verification proof outline of a program consists of using intermediate assertions at each step of the program to show that if the precondition is true before the execution of the program then the postcondition is true after execution of the program. Each intermediate assertion states what properties the variables and communication sequences must possess at each step of the program.

A faulty program violates at least one of the intermediate assertions used in the proof outline. Therefore, it is feasible to embed one or more of the intermediate assertions into the operational environment as an executable assertion to catch faulty behavior. This forms the basis of the procedure for translating a proof outline to fault-tolerant constraints.

The procedure for translating a proof outline to fault-tolerant constraints proceeds as follows:

1. **Delete all assertions that use only local variables.** This leaves only assertions that use global auxiliary variables.

2. **Choose a subset of the assertions that use global auxiliary variables.** It is this subset that will be embedded into the operational environment as an executable assertion. These assertions will become the progress and feasibility constraints.

3. **Communicate the variables used in the subset of assertions chosen is step two.** Communication must be done in such a way that it is possible to test for consistency. The communication of variables used in the subset of assertions chosen in step two should attempt to minimize
extra communication steps.

In step two, the choice of assertions is influenced by several criteria. The first criterion is based on that auxiliary variables are generally used to record part of the history of communication. Assertions that use only local variables could limit the operational fault-tolerant environment to having each node check only itself or limit the search for faults to only node faults as opposed to both node and communication link faults.

The second criterion is based on avoiding having processes do only self-checking. It is desirable that a process sends the values of the variables required for determining truth of the verification assertions chosen as executable assertions to other processes. Therefore, for a process \( i \), there will be several other processes, determining whether the values of variables in process \( i \) satisfy the verification assertion. In other words, different processes will execute the same executable assertions. Because of the possibility of faulty processes, it is desirable to send the values to the different processes on two different communication paths. Thus, consistency is necessary to ensure that processes executing the same executable assertion has a consistent view of the data.

The third criterion is based on the desire to minimize extra communication steps described in step three. For a process \( i \) in a concurrent program, let \( H_i \) denote the sequence of processes in which process \( i \) communicates and let \( H_i' \) denote the sequence of processes in which process \( i \) communicates in a concurrent program in with instrumented assertions. Then if we have that for any sequence that the function labelled length will return the number of elements in a sequence, then the attempt to minimize extra communication steps is mathematically described as minimizing for all \( i \) the following equation:

\[
\text{length}(H_i') - \text{length}(H_i)
\]

We now turn to the specific application of parallel matrix iterative relaxation. We will derive several executable assertions from part of the assertion Loop\(_i\). Loop\(_i\) uses the following auxiliary variables \( c_u_i, p_u_i, u_{\text{max}} \) and \( u_{\text{min}} \). It will be shown that it is only necessary to explicitly communicate \( c_u_i \). Since, all processes require the use of the auxiliary variables \( u_{\text{max}}, u_{\text{min}} \) remain constant throughout execution then it is assumed that each of these processes has a copy of both \( u_{\text{max}} \) and \( u_{\text{min}} \). Therefore, in this problem, it sufficient to send the value of \( c_u_i \) computed in each stage to other processes.

Consistency as defined in Section IV is general. As discussed before, the strict notion of time \( T \) in Section IV can be relaxed significantly in specific applications. Since, it is only necessary to know the stage a particular value of \( c_u_{\text{node}} \) was computed, then the time at which a particular value of \( c_u_{\text{node}} \) was computed is not necessary. For this application, the auxiliary variables
to be communicated for each process $i$ is $cu_i$ and $pu_i$. It can be shown that the communication of $pu_i$ can be expressed in terms of $cu_i$. Hence, we are concerned in explicitly communicating $cu_i$.

In the notation, $cu_i$ implies the variable type. We are also not concerned with the specific path in which auxiliary variables traverse. We are only concerned in distinguishing different paths.

One possible way of communicating variables based on the above criteria is to have each process labelled node such that $\Lambda_{node,k} = 1$ in stage $k$ send the value of $cu_{node}$ computed in stage $k$ (i.e. $u_{node}^{(k)}$) to the processes in the labelled North, South, West, East in stage $k+1$. Each of these processes will then send $cu_{node}$ in stage $k+2$ to the processes involved direction in the computation of $u_{north}, u_{south}, u_{west}, u_{east}$. Mathematically this communication can be described as follows:

**Definition 6.1**: Define $node_{h_1}^k$ for $k > 0$ as follows:

$$node_{h_1}^k = \begin{cases} <\text{North}, cu_{north}, N>, <\text{South}, cu_{south}, S>, <\text{East}, cu_{east}, E>, <\text{West}, cu_{west}, W> \end{cases}$$

where $N, S, E, W \in P$ and $P$ is a set of path directions. and define $node_{h_2}^k$ for $k > 1$ as follows:

$$node_{h_2}^k = node_{h_1}^k \cup \begin{cases} \text{North}_h^k \cup \text{South}_h^k \cup \text{East}_h^k \cup \text{West}_h^k \end{cases}$$

where $N, S, E, W$ are used to help distinguish the different paths that $cu_i$ is communicated on.

Figure 6.1 illustrates part of the communication.
node $h^2_k$ denotes the values of auxiliary variables received by the process node in the current stage that were computed two stages ago. node $h^1_k$ denotes the values of auxiliary computed one stage ago and received in the current stage. A process labelled node does not receive auxiliary variables in stage $k$ if it is not computing a new value for $u_{node}$, i.e. $L_{node,k} = 1$. 

**Figure 6.1(a):** Communication of Auxiliary Variables of node in stage $k+1$

**Figure 6.1(b):** Communication of Auxiliary Variables of node in stage $k+2
It will now be shown that for each process labelled node that computes a new value for $u_{\text{node}}$ in stage $k$, that this communication results in communication of auxiliary variables to four processes on two vertex disjoint paths. Lemmas 6.1 and 6.2 are lemmas stating certain equalities. The purpose of stating these lemmas is to simplify the proof of Theorem 6.1, which states that a process can send values of auxiliary variables to four processes on two vertex disjoint paths. This is illustrated in Figure 6.1.

**Lemma 6.1:** For any process labelled node, the following is true:

- $\text{North}_{\text{West}} = \text{West}_{\text{North}}$
- $\text{North}_{\text{East}} = \text{East}_{\text{North}}$
- $\text{South}_{\text{West}} = \text{West}_{\text{South}}$
- $\text{South}_{\text{East}} = \text{East}_{\text{South}}$

**Proof:** This proof will show that $\text{North}_{\text{West}} = \text{West}_{\text{North}}$. The other equalities are shown in a similar manner. From Section V, we have the following:

- $\text{North} = (\text{node div } \sqrt{Q} - 1, \text{node mod } \sqrt{Q})$
- $\text{West} = (\text{node div } \sqrt{Q}, \text{node mod } \sqrt{Q} - 1)$

We then have that

- $\text{north} = (\text{node div } \sqrt{Q} + 1) * \sqrt{Q} + \text{node mod } \sqrt{Q}$
- $\text{west} = (\text{node div } \sqrt{Q}) * \sqrt{Q} + \text{node mod } \sqrt{Q} - 1$

Therefore, we have that:

- $\text{North}_{\text{West}} = (\text{node div } \sqrt{Q} + 1, \text{node mod } \sqrt{Q} - 1)$
- $\text{West}_{\text{North}} = (\text{node div } \sqrt{Q} + 1, \text{node mod } \sqrt{Q} - 1)$

We can then conclude that $\text{North}_{\text{West}} = \text{West}_{\text{North}}$. □

**Lemma 6.2:** For a process labelled node, we have that:

- $\text{East}_{\text{West}} = \text{node}$
- $\text{West}_{\text{East}} = \text{node}$
- $\text{North}_{\text{South}} = \text{node}$
- $\text{South}_{\text{North}} = \text{node}$

**Proof:** This proof will show that $\text{East}_{\text{West}} = \text{node}$ is true. The other equalities are shown in a
similar manner. From Section V, we have the following:

\[ \text{East} = (\text{node} \ \text{div} \sqrt{Q}, \text{node} \ \text{mod} \sqrt{Q} + 1) \]

From Section V, we have that

\[ \text{east} = (\text{node} \ \text{div} \sqrt{Q}) \ * \sqrt{Q} + \text{node} \ \text{mod} \sqrt{Q} + 1 \]

Therefore, we have that:

\[ \text{East}_{\text{West}} = (\text{node} \ \text{div} \sqrt{Q}, \text{node} \ \text{mod} \sqrt{Q}) \]

\[ \text{(node } \ \text{div} \sqrt{Q}, \text{node } \ \text{mod} \sqrt{Q}) \text{ is equal to node. Hence, we have that East}_{\text{West}} = \text{node}. \]

Theorem 6.1 shows that a process can send values of auxiliary variables to four processes on two vertex disjoint paths.

**Theorem 6.1:** For a process labelled node and \( k > 1 \),

\[ \exists t_1, t_2 \in \text{North}_k \text{West}_2 \]

\[ \exists t_1, t_2 \in \text{North}_k \text{East}_2 \]

\[ \exists t_1, t_2 \in \text{South}_k \text{West}_2 \]

\[ \exists t_1, t_2 \in \text{South}_k \text{East}_2 \]

such that

\[ t_1 = <\text{node}, v_1, \text{direction}_1> \]

\[ t_2 = <\text{node}, v_2, \text{direction}_2> \]

**Proof:** We will examine the case of \( \text{North}_k \text{West}_2 \). The other cases are symmetric. From Lemma 6.1, we know that \( \text{North}_{\text{West}} = \text{West}_{\text{North}} \). For a process labelled node, Lemma 6.2 showed that \( \text{North}_{\text{South}} = \text{node} \). Therefore, from Definition 6.1, we have that there is a \( t_1 \in \text{North}_1 \) such that \( t_1 = <\text{node}, v_1, S> \). Similarly, there is a \( t_2 \in \text{West}_1 \) such that \( t_2 = <\text{node}, v_2, E> \). Since, \( \text{North}_{\text{West}} = \text{West}_{\text{North}} \) and by applying Lemma 6.2, we can show that \( \text{West}_{\text{North}} \text{South} = \text{West} \) and that \( \text{North}_{\text{West}} \text{East} = \text{North} \), then using Definition 6.1, we can conclude that there exists \( t_1, t_2 \in \text{North}_k \text{West}_2 \) such that

\[ t_1 = <\text{node}, v_1, S> \]

\[ t_2 = <\text{node}, v_2, E> \]
The communication of auxiliary variables as defined by Definition 6.1 allows for a process labelled node to receive values on two vertex disjoint paths. A corollary to Theorem 6.1 shows that for a process labelled node that it receives auxiliary variables from four processes on two vertex disjoint paths.

**Corollary 6.1:** For a process labelled node, there exists \( t_1, t_2 \in \text{node}^4 \) such that

\[
\begin{align*}
  t_1 &= \langle \text{NorthWest}, v_1, \text{direction}_1 \rangle \\
  t_2 &= \langle \text{NorthWest}, v_2, \text{direction}_2 \rangle
\end{align*}
\]

A similar statement can be said for NorthEast, SouthWest, and SouthEast.

**Proof:** Proof is similar to Theorem 6.1. \( \Box \)

It is assumed that a processor only tests values of auxiliary variables that it receives from two vertex disjoint paths. To achieve consistency, each process testing for a process labelled node “hears” hears the same version. This is mathematically defined as follows:

**Definition 6.2:** If there exists \( t_1, t_2 \in \text{node}^4 \), where

\[
\begin{align*}
  t_1 &= \langle I, cu_i, \text{direction}_1 \rangle \\
  t_2 &= \langle J, cu_j, \text{direction}_2 \rangle
\end{align*}
\]

and \( I = J \), but \( cu_i \neq cu_j \), then the system is said to be inconsistent else the system is said to be consistent.

We now discuss the efficiency of the consistency test of Definition 6.2 under a faulty system.

**Definition 6.3:** Local Fault Group: \( \Omega_{\text{node}} = \{ P_i | a_{i,j} \neq 0, j = 1, \ldots, N \} \).

Therefore, \( \Omega_{\text{node}} \) is the set of processes involved directly in the computation of \( u_{\text{node}} \) i.e. \( \Omega_{\text{node}} \) includes the processes labelled North, South, West, East.

**Definition 6.4:** Local Test Group: \( K_{\text{node}} = \{ \text{NorthEast}, \text{NorthWest}, \text{SouthWest}, \text{SouthEast} \} \).

**Definition 6.5:** Test Group: \( T_{\text{node}} = \Omega_{\text{node}} \cup K_{\text{node}} \).

**Definition 6.6:** Communication Consistency Conditions: Let the communication consistency conditions be defined as follows:
V1 For any process labelled node, any two non-faulty processes labelled I and J, where I, J ∈ K\textsubscript{node}, ∃ t\textsubscript{1}, t\textsubscript{2} in I\textsubscript{h}\textsubscript{k}, J\textsubscript{h}\textsubscript{k} respectively, and

\[ t\textsubscript{1} = \langle \text{node}, cu\textsubscript{i}, \text{direction} \rangle \]
\[ t\textsubscript{2} = \langle \text{node}, cu\textsubscript{j}, \text{direction} \rangle \]

then cu\textsubscript{i} = cu\textsubscript{j} or else an inconsistent system is detected.

V2 If a process labelled node is non-faulty in stage k, then for any non-faulty process labelled I, where I ∈ K\textsubscript{node} and t\textsubscript{1} = \langle \text{node}, v, \text{direction} \rangle, then v = cu\textsubscript{i} or else an inconsistent system is detected.

Condition V1 states that any two non-faulty processes in K\textsubscript{node} should receive the same values of the auxiliary variable cu\textsubscript{node}. However, these values may or may not be equal to the value computed in process node. V2 ensures the value received by the processes in K\textsubscript{node} is the correct value. It will now be shown using that the consistency test of Definition 6.2 achieves the communication consistency conditions. Lemma 6.3 shows that the consistency test implies V1 for one fault. Lemma 6.4 shows that the consistency test implies V2 for one fault. Theorem 6.2 summarizes these results.

Lemma 6.3: For a process labelled node, the consistency condition of Definition 6.2 satisfies V1 when the maximum number of faulty processors in Test\textsubscript{node} is one.

Proof: If the number of faults is zero, then V1 is trivially true. Assume that there is one faulty process in Test\textsubscript{node}. There are two cases to consider. The first case is that the faulty process is in K\textsubscript{node}. V1 is trivially true, since the consistency test must be satisfied for non-faulty processes. The second case to consider is when the faulty process is in Ω\textsubscript{node}. From Theorem 6.1, we have that there exists two vertex disjoint paths from the process labelled node to the processes in Test. If the faulty process in Ω\textsubscript{node} changes the value of an incoming auxiliary variable from process node, then since there exists two vertex disjoint paths from the process labelled node to the processes in Test and there is only one faulty process, then one of the processes in Test will receive two different values. From Definition 6.2, we have that the consistency test for the process receiving two different values will flag the system as being inconsistent. □

Lemma 6.4: For a process labelled node, the consistency condition of Definition 6.2 satisfies V2 when the maximum number of faulty processors in Test\textsubscript{node} is one.

Proof: If the number of faults is zero, then V2 is trivially true. Assume that there is one faulty process in Test\textsubscript{node}. There are two cases to consider. The first case is that the faulty process is in
$K_{\text{node}}$. V2 is trivially true, since the consistency test must be satisfied for non-faulty processes.

The second case to consider is when the faulty process is in $\Omega_{\text{node}}$. From Theorem 6.1, we have that there exists two vertex disjoint paths from the process labelled node to the processes in Test. If the faulty process in $\Omega_{\text{node}}$ changes the value of an incoming auxiliary variable from process node, then since there exists two vertex disjoint paths and there is only one faulty process, then one of the processes in Test can detect the discrepancy using the consistency test. Since there is only one fault, the rest of the non-faulty processes in Test will receive the same value from both paths. □

**Theorem 6.2:** The consistency condition achieves communication consistency in the presence of one fault in each test group.

**Proof:** The immediate result of Lemmas 6.3 and Lemmas 6.4. □

Once a processor knows that it has a consistent view of the auxiliary variables, then it is possible to test whether the auxiliary variables satisfy the assertion Loop$_i$. One component of the feasibility test is derived from Theorem 5.3. Theorem 5.3 states that for each $u_{\text{node}}^{(k)}$ that $u_{\text{min}} \leq u_{\text{node}}^{(k)} \leq u_{\text{max}}$. From Theorem 6.1, we have that for a process labelled node that the processes labelled NorthWest, NorthEast, SouthWest, SouthEast receive values of the auxiliary variable $c_{\text{node}}$ on two vertex disjoint paths. If it is assumed that each process knows the values of $u_{\text{min}}$ and $u_{\text{max}}$, then for each such value received it can be tested whether each value satisfies the conditions of Theorem 5.2.

One component of the progress test is derived from Theorem 5.2. The auxiliary variable $c_{u_i}$ denotes the current value of $u_i$, while $p_u_i$ denotes the previous value of $u_i$. Each value computed for $u_i$ is sent to other processes. Hence, as each new value of $u_i$ arrives at a checking process, the old value is now known to be the value of the auxiliary variable $p_{u_i}$. Therefore, one component of the progress test is to check whether $c_{u_i}, p_{u_i}$ satisfy the conditions of Theorem 5.2.

Another component of the progress test is derived from testing whether $r_1^{(k)} = 0$. The assertion that $r_1^{(k)} = 0$ in stage k implies Theorem 6.3. The significance of this theorem is that it allows for testing of $r_1^{(k)} = 0$ in terms of the current and previous value of $u_i$ and the consecutive sequence values of $u_i$, where i is a process directly involved in the computation of $u_i$. 
Theorem 6.3: Consider a sequence of \( u_i^{(k)} \)'s satisfying the conditions of Theorem 5.2. From the perspective of a processor \( P_i \) calculating \( u_i \), a candidate intermediate result \( u_i^{(k)} \) with a non-zero coefficient \( a_{i,j}, 1, i \in \{1, 2, \ldots, Q\} \) must satisfy the following properties

If \( u_i^{(0)} = u_{\text{min}}, i \in \{1, 2, \ldots, Q\} \), then

\[
u_i^{(k+2)} \leq u_i^{(k)} - \frac{1}{a_{i,l}} \left[ a_{l,i} (u_i^{(k+1)} - u_i^{(k-1)}) \right], \quad k \in \{0, 1, 2, \ldots\}
\]

Proof: Consider two successive iterations \( k \) and \( k+2 \). Let \( r_i^{(k)} \) denote the residual value of \( u_i \) in stage \( k \) and let \( r_i^{(k+2)} \) denote the residual value of \( u_i \) in stage \( k+2 \). If \( r_i^{(k)} = 0 \) then this implies that

\[
\sum_{j \neq l} a_{ij} u_j^{(k-1)} + a_{il} u_i^{(k)} = v_l
\]

and \( r_i^{(k+2)} = 0 \) implies that

\[
\sum_{j \neq l} a_{ij} u_j^{(k+1)} + a_{il} u_i^{(k+2)} = v_l
\]

This in turn implies that

\[
\sum_{j \neq l} a_{ij} u_j^{(k-1)} + a_{il} u_i^{(k)} = \sum_{j \neq l} a_{ij} u_j^{(k+1)} + a_{il} u_i^{(k+2)}
\]

\[
a_{il} [u_i^{(k+2)} - u_i^{(k)}] = \sum_{j \neq l} a_{ij} u_j^{(k-1)} - \sum_{j \neq l} a_{ij} u_j^{(k+1)}
\]

\[
u_i^{(k+2)} - u_i^{(k)} = \frac{1}{a_{il}} \left[ \sum_{j \neq l} a_{ij} u_j^{(k-1)} - \sum_{j \neq l} a_{ij} u_j^{(k+1)} \right]
\]

\[
u_i^{(k+2)} - u_i^{(k)} = \frac{1}{a_{il}} \left[ \sum_{j \neq l} a_{ij} u_j^{(k+1)} + a_{il} u_i^{(k+1)} - (\sum_{j \neq l} a_{ij} u_j^{(k-1)} + a_{il} u_i^{(k-1)}) \right]
\]

Since \( u_i^{(k-1)} \geq u_i^{(k)} , k \in \{1, 2, \ldots\} \)

\[
\leq \frac{1}{a_{il}} (a_{il} u_i^{(k+1)} - a_{il} u_i^{(k-1)}) \]

Theorem 6.3 shows that for a process labelled node, that the check for the condition \( r_{\text{node}}^{(k)} = 0 \) could be replaced by

\[
u_{\text{node}}^{(k+2)} \leq u_{\text{node}}^{(k)} - \frac{1}{a_{\text{node,node}}} \left[ a_{\text{node,node}} (u_{\text{node}}^{(k+1)} - u_{\text{node}}^{(k-1)}) \right], \quad k \in \{0, 1, 2, \ldots\}
\]
where the process labelled l is a process involved directly in the solution for \( u_{\text{node}} \). Therefore, we have that \( l \in \{ \text{North, South, West, East} \} \). A corollary of Theorem 6.3 is the following:

**Corollary 6.2:** A candidate intermediate result \( u_i^{(k)} \) with nonzero coefficient \( a_{i,l} \), from the perspective of processor calculating \( u_i^{(k)} \), \( l \in \{ 1, 2, \ldots, Q \} \), must satisfy the following properties.

If \( u_i^{(0)} = u_{\text{min}}, \ i \in \{ 1, 2, \cdots, Q \} \), then

\[
    u_i^{(k+2)} \geq -\frac{1}{a_{i,l}} \left[ \sum_{j \neq i, j \neq i} a_{i,j} u_{\text{max}} + a_{i,l} u_i^{(k+1)} - v_l \right]
\]

Similarly if \( u_i^{(0)} = u_{\text{max}}, \ i \in \{ 1, 2, \cdots, Q \} \), then

\[
    u_i^{(k+2)} \leq -\frac{1}{a_{i,l}} \left[ \sum_{j \neq i, j \neq i} a_{i,j} u_{\text{min}} + a_{i,l} u_i^{(k+1)} - v_l \right]
\]

**Proof:** Immediate from the conditions of Theorem 5.2 and the proof of Theorem 6.3 \( \square \).

Lemma 6.7 and Theorem 6.4 show that for a process labelled node, that the method of communicating auxiliary variables as defined in Definition 6.1 allows for the processes labelled NorthWest, NorthEast, SouthWest, SouthEast to test whether \( u_{\text{node}} \) satisfies Theorem 6.3 and Corollary 6.2. In other words, it is shown that the processes involved directly in the computation of \( u_{\text{node}} \) communicate values of their auxiliary variables according to Definition 6.1 to the processes labelled NorthWest, NorthEast, SouthWest, SouthEast. This allows for these processes to check whether the values of \( c u_{\text{node}} \) and \( p u_{\text{node}} \) satisfy Theorem 6.3.

**Lemma 6.7:** For a process labelled node, the following is true:

\[
\begin{align*}
(\text{NorthWest})_{\text{East}} &= \text{North} \\
(\text{NorthEast})_{\text{West}} &= \text{North} \\
(\text{SouthWest})_{\text{East}} &= \text{South} \\
(\text{SouthEast})_{\text{West}} &= \text{South} \\
(\text{EastNorth})_{\text{South}} &= \text{East} \\
(\text{EastSouth})_{\text{North}} &= \text{East} \\
(\text{WestNorth})_{\text{South}} &= \text{West} \\
(\text{WestSouth})_{\text{North}} &= \text{West}
\end{align*}
\]

**Proof:** The proof is similar to that used in Lemmas 6.1 and Lemmas 6.2. \( \square \)
Theorem 6.4: For a process labelled node, the processes labelled NorthWest, NorthEast, SouthWest, SouthEast receive values of the auxiliary variable $c_{ui}$, where the process labelled $l$ is a process involved directly in the solution for $u_{node}$.

Proof: Trivial from Definition 6.2 and Lemma 6.7.

As will be shown in Section VII all the constraint predicates derived to this point will find about a high percentage of all faults with a fault latency of one, but not all faults. Therefore, in order to ensure more accurate fault detection, another verification proof assertion was chosen in order to derive more executable assertions. The chosen assertion is the postcondition, which includes that for each $i$ and if the stopping point is selected to be the iteration $K$, then it is the case that $|u_i^{(K)} - u_i^{(K+2)}| < \epsilon$. This can be used to derive Theorem 6.5.

Theorem 6.5: At the end of the final iteration $K$, the final result $u_i^{(K)}$ satisfies the following relations

If $u_i^{(0)} = u_{min}, \ i \in \{1, 2, \cdots, Q\}$, then

$$u_i^{(K)} \geq \frac{1}{a_{i,i}} \left[ \sum_{j \neq i}^Q a_{i,j} u_j^{(K)} - v_i \right] \geq u_i^{(K)} - \epsilon$$

and if $u_i^{(0)} = u_{max}, \ i \in \{1, 2, \cdots, Q\}$, then

$$u_i^{(K)} \leq \frac{1}{a_{i,i}} \left[ \sum_{j \neq i}^Q a_{i,j} u_j^{(K)} - v_i \right] \leq u_i^{(K)} + \epsilon$$

Proof: Consider the final iteration at $K + 2$. Assume that $u_i^{(0)} = u_{min}, \ i \in \{1, 2, \cdots, Q\}$.

$$u_i^{(K+2)} = -\frac{1}{a_{i,i}} \left[ \sum_{j \neq i}^Q a_{i,j} u_j^{(K+1)} - v_i \right]$$

Since $u_i^{(K)} - u_i^{(K+2)} < \epsilon$,

$$\epsilon > u_i^{(K)} - \frac{1}{a_{i,i}} \left[ \sum_{j \neq i}^Q a_{i,j} u_j^{(K+1)} - v_i \right]$$

The case when $u_i^{(0)} = u_{max}, \ i \in \{1, 2, \cdots, Q\}$ is symmetric.

The test of Theorem 6.5 can be done for process labelled node by the processes labelled NorthWest, NorthEast, SouthWest, SouthEast, by ensuring that the communication of auxiliary
variables as defined in Definition 6.2 is done in an extra stage. The communication consistency conditions ensure that these values are reliably communicated.

To summarize these are the following progress and feasibility tests to be done for the values computed in each stage $k$ for each process node:

Feasibility (Derived from Theorem 5.3):

$$u_{\text{min}} \geq u_{\text{node}}^{(k)} \geq u_{\text{max}}$$

Progress 1 (Derived from Theorem 5.2):

$$u_{\text{node}}^{(k)} \geq u_{\text{node}}^{(k+2)}$$

and

Progress 2: (Derived from Theorem 6.3)

$$u_{\text{node}}^{(k+2)} \leq u_{\text{node}}^{(k)} - \frac{1}{a_{\text{node, node}}^{(k)}} a_{\text{node, l}}^{(k)} (u_l^{(k+1)} - u_l^{(k-1)})$$

Theorem 6.1 shows that for a process labelled node, there are four processes labelled NorthWest, NorthEast, SouthWest, and SouthEast that receive the value of $c u_{\text{node}}$ in stage $k$ in stage $k+2$ on two vertex disjoint paths. It has also been described how to determine $p u_{\text{node}}$ from communicated values of $c u_{\text{node}}$. In order for the processes labelled NorthWest, NorthEast, SouthWest, and SouthEast to do the test as described by Progress 2, they must receive values of $c u_l$, where $l$ is a process involved directly in the solution for $u_{\text{node}}$. This is shown in Theorem 6.4.

The above tests are done for each stage $k$. However, Theorem 6.5 describes a feasibility test to be done at the end of the program, after an extra communication step.

VII. ERROR COVERAGE MODELING FOR MATRIX ITERATIVE ANALYSIS

One metric that can be used in determining the effectiveness of a testing scheme is the expected coverage of errors.

To model the error coverage we have two options. One is to generate a large number of test cases, introduce random errors, and measure the coverage. This is unattractive since it does not show which tests contribute most to the error coverage and which tests are redundant; nor does it provide much confidence in the coverage unless a large number of tests is run. The alternative is to use an analytical model. A probability model is chosen.

In general, to utilize a probability model, both a prior and posterior probability distribution of errors must be chosen. Here, however, we are only interested in the coverage of the constraint predicate given that an error has occurred. This restricts the number of distributions to be
considered to the distribution of errors given that an error has occurred. The fault model for application data errors can be chosen from some prior distribution appropriate to either the hardware faults expected, or as a function of the application itself.

**Model Specifications:** Define the following events corresponding to the three classes of test bases:

- Event that a result \( U^{(k)}_{ij} \) satisfies feasibility test \( i \).
- Event that a result \( U^{(k)}_{ij} \) satisfies progress test \( i \).
- Event that an error has occurred.

The absence of a particular subscript/superscript indicates that the event holds for an aggregation of the individual events. Thus the following events denoting the intersection of each class of error coverage are true when all of the tests of a particular class are true:

\[
\begin{align*}
{jF}^{(k)} &= \bigcap_i j{F}^{(k)}_i \\
{jP}^{(k)} &= \bigcap_i j{P}^{(k)}_i
\end{align*}
\]

The absence of the superscript \( (k) \) indicates that the event is stationary over all steps \( k \) as in \( j{F}_i \).

The meaning will be made clear in each individual context.

We are interested in the probability that the tests find an error if it has occurred. This probability is given by the following:

\[
\Pr(F \cap P|E) = \Pr(F \cup \bar{P}|E) = \Pr(F|E) + \Pr(P|E) - \Pr(FP|E) - \Pr(F|E) - \Pr(P|E) + \Pr(FP|E)
\]

To model the type of errors that can occur, we can choose either a discrete density function or continuous distribution function. In the former case, the discrete nature results from the discreteness of the computer word used to contain the result. The probability function of the random variable denoting the actual value of the result is the hypergeometric distribution. Assume that the range of values that a computer word can hold is \( \{-L, -L + 1, \ldots, L - 1, L\} \). Then the conditional probability of the correct result (and any particular result) is \( 1/2L \). For any reasonable size computer word length, this value is nearly zero, which is exactly what we want. In the latter case, consider the choice of the distribution of errors from some well defined distribution such as the normal. The mean of the distribution is chosen to be equal to the actual correct value of the calculation. The choice of variance can control whether erroneous values are in close proximity to the correct value or are more widely scattered. Another possibility is to choose the conditional
density function to be uniform over \([-L, L]\) as in the discrete case above. In both cases, since the distribution is continuous, the conditional probability of the correct result again is zero. We are not limited to such standard distributions. If \textit{a priori} knowledge is available concerning the probability of faulty behavior, the model can be adjusted to accommodate.

To show how the modeling works, consider as an example a feasibility predicate called predicate \(F^A\), which states that a certain variable must be positive, and its corresponding feasibility event \(iF_1^{(k)}\). Assume that there is only one component of the solution, so the leading superscript is dropped. Similarly, since the final value is the only one considered, the step number is dropped. If we model the errors as coming from a discrete random distribution with the random variable \(E_b\), then the conditional probability of detecting an error is given by

\[
Pr(F|E) = Pr(E_b < 0) = \frac{L}{2L} = 0.50
\]

This shows, as we already suspected, that the predicate has a low error coverage - only 50%.

In modeling the errors as a normally distributed random variable \(E_c\), we assume that the mean is chosen to be the actual correct value of the area \(\delta\). For simplicity, let the variance \(\sigma = 1\). The probability of error detection is given by:

\[
Pr(F|E) = Pr(E_c < 0) = Pr(E_c - \delta < -\delta) = Pr(Z < -\delta)
\]

where \(Z\) has the standard normal distribution. If we choose \(\delta\) to be, say, 2, then

\[
Pr(Z < -2) = .0667
\]

or just 7% error coverage. Clearly under this model, the predicate \(F^A\) is virtually useless.

In constraint predicate analysis, it is usually true that one predicate significantly narrows the range of acceptable values, and that additional predicates only make small contributions. This type of modeling can help eliminate predicates with little usefulness to increase run time efficiency at the cost of lower error coverage.

### 7.1 Feasibility

Modeling of the error coverage of the feasibility constraint derived from Theorem 5.3 is straightforward. The feasibility events \(F_i, i = 1, 2\) are defined by

\(iF_1^{(k)}\): Event that \(u_j^{(k)} \geq u_{\text{max}}\) for \(j\) and \(k\) under test and

\(iF_2^{(k)}\): Event that \(u_i^{(k)} \leq u_{\text{min}}\) for the \(i\) and \(k\) under test.

Since these constraints are stationary

\[
Pr[iF^i] = Pr[iF_1^{(k)}] \text{ for all } j \in \{1, 2, \ldots, Q\}, \ k \in \{0, 1, 2, \ldots\}, \text{ and } i = 1, 2.
\]
The cumulative distribution function of the appropriate model yields the coverage probabilities for all steps of the problem.

The corresponding probabilities are given by

\[ P[r^j_F_1] = P[u_j^{(k)} \geq u_{\text{max}}] \]

\[ P[r^j_F_2] = P[u_j^{(k)} \leq u_{\text{min}}] \]

and

\[ P[r^j_F] = P[u_{\text{max}} \leq u_j^{(k)} \leq u_{\text{min}}] \]

### 7.2 Progress

Progress is modeled by considering the global convergence properties of the solution and extrapolating this behavior back to each individual case. [Ames77] gives a bound on the rate of error reduction in a matrix iterative solution.

The first step in this analysis is to determine the spectral radius \( \rho_G \) of the iteration matrix \( G \). It is possible to determine a close approximation to \( \rho_G \) by examining the ratio of the norm of the error vectors between iterations. However, \( \rho_G \) can also be calculated directly from \( G \), while in general is not feasible due to the size of \( G \), for analytical purposes is sufficient. The matrix \( A \) is decomposed into \( A = B + D + C \) with \( B \) lower triangular, \( C \) upper triangular, and \( D \) diagonal \( Q \times Q \) matrices. Thus \( G \) becomes,

\[ G = (D + B)^{-1}[-C] \]

If the complex eigenvalues of \( G \) are given by \( \lambda_i \) for \( i \in \{1, \ldots, Q\} \), then

\[ \rho_G = \max_{i \in \{1, \ldots, Q\}} |\lambda_i| \]

Let the error at step \( k \) of the computation be denoted by the vector \( \varepsilon^{(k)} = |u^k - u| \) for \( k \in \{0, 1, 2, \ldots\} \). For a stationary linear iteration matrix \( G \) of the form considered in Theorem 5.1, \( \|\varepsilon^{(k)}\| = \|G^k \varepsilon^{(0)}\| \) where \( \|\varepsilon^{(k)}\| \) is the spectral norm of \( \varepsilon^{(k)} \). If \( \rho_G < 1 \), for \( k \) sufficiently large, \( \|G^k\| = (\rho_G)^k \) where \( \|G\| \) is the spectral norm of \( G \). Combining these results yields [Ames77]

\[ \|\varepsilon^{(k)}\| \leq \|G^k\|\|\varepsilon^{(0)}\| \]

Thus, for large \( k \) the ratio \( \|\varepsilon^{(k+1)}\|/\|\varepsilon^{(k)}\| \) averages to \( \rho_G \). Therefore, on the average, the error decreases by a factor of \( \rho_G \) at each step in the iteration. For modeling purposes, the final result is known. Assuming that each individual point behaves according to global convergence and using Theorem 5.2, treatment of an arbitrary point, \( i' \), yields the following relation...
If $u_i^{(0)} = u_{min}$,

$$\rho_G^k(u_{min} - u_i^\cdot) \geq u_i^{(k)} - u_i^\cdot, \text{ for } k \in \{0, 1, 2, \ldots \} \quad (7.1)$$

where $u_i^\cdot$ is the correct final result of the calculation. A similar result holds if $u_i^{(0)} = u_{max}$. The events $P_i$ that derive the component of progress derived from Theorem 5.2 are the following:

If $u_i^{(0)} = u_{min}$,

$i' P_{1a}^{(k)}$ is the event $u_i^{(k)} \leq \rho_G^k(u_{min} - u_i^\cdot) + u_i^\cdot$, $k \in \{0, 1, 2, \ldots \}$

If $u_i^{(0)} = u_{max}$,

$i' P_{1b}^{(k)}$ is the event $u_i^{(k)} \geq u_i^\cdot - \rho_G^k(u_i^\cdot - u_{max})$, $k \in \{0, 1, 2, \ldots \}$

The progress components of Theorem 6.3 and Corollary 6.2, like the progress component derived from Theorem 5.2, can either be determined from a trace or by analytic means. For the analytic model, we apply Theorem 6.3 with the assumption that each $u_i$ proceeds toward the solution $U$ at the same convergence rate. Again this is not the case in the actual solution but does effectively display the aggregate behavior of the solution. While Theorem 6.3 and Corollary 6.2 are directly implementable as a constraint predicate, to use it in the model requires additional simplification. The ratio $a_{i,j}/a_{i,i}$ is aggregated by replacing it with the ratio $R = \frac{u_i^{(k)}}{nu_{i}^{(k-1)}}$ where $n$ is the average number of nonzero $a_{i,j}$'s. Using (7.1) as an equality and substituting into Theorem 6.3 yields the events $i' P_{2a}^{(k+2)}$ and $i' P_{2b}^{(k+2)}$

If $u_i^{(0)} = u_{min}$, $i' P_{2a}^{(k+2)}$ is the event,

$$u_i^{(k+2)} \leq \rho_G^k(u_{min} - u_i^\cdot)$$

$$- R(\rho_G^{k+1}(u_{min} - u_i^\cdot) - \rho_G^{k-1}(u_{min} - u_i^\cdot)), \text{ for } k \in \{2, 3, 4, \ldots \}$$

and if $u_i^{(0)} = u_{max}$, $i' P_{2b}^{(k+2)}$ is the event,

$$u_i^{(k+2)} \geq \rho_G^k(u_{min} - u_i^\cdot)$$

$$- R(\rho_G^{k+1}(u_{min} - u_i^\cdot) - \rho_G^{k-1}(u_{min} - u_i^\cdot)), \text{ for } k \in \{2, 3, 4, \ldots \}.$$

Letting the mean of the $v_i$'s be $\bar{v}$, substitution into Corollary 6.2 yields the events $i' P_{3a}^{(k+2)}$ and $i' P_{3b}^{(k+2)}$.

If $u_i^{(0)} = u_{min}$, $i' P_{3a}^{(k+2)}$ is the event,

$$u_i^{(k+2)} \geq - R \left[ \sum_{j \neq i}^Q \frac{u_{max} + \rho_G^{k+1}(u_{min} - u_i^\cdot) + u_i^\cdot - \bar{v}}{n} \right], \text{ for } k \in \{2, 3, 4, \ldots \};$$
and if \( u_i^{(0)} = u_{\text{max}} \), \( i, \beta_{3a}^{(k+2)} \) is the event,
\[
u_i^{(k+2)} \leq - R \left[ \sum_{j \neq i} u_{\text{min}} + u_i - \rho_G^{k+1}(u_i - u_{\text{min}}) - v \right], k \in \{2, 3, 4, \ldots \}.
\]

With these relationships defined, the modeling can proceed. Choose as an example problem the matrix shown in Figure 7.1 that satisfies the conditions of Theorem 5.1. The spectral radius \( \rho_G \) of the corresponding iteration matrix \( G \) is 0.4775. It is easily verified that \( u_{\text{min}} = 1.00 \) (from rows 5 or 6 of \( A \)) and \( u_{\text{max}} = .057 \) (from row 4 of \( A \)).

\[
A = \begin{bmatrix}
4.87 & -1.28 & 0 & -1.37 & 0 & 0 & 0 & 0 \\
-1.28 & 5.62 & -1.78 & 0 & -1.37 & 0 & 0 & 0 \\
0 & -1.78 & 6.87 & 0 & 0 & -1.37 & 0 & 0 \\
-1.37 & 0 & 0 & 5.37 & -1.28 & 0 & -1.62 & 0 \\
0 & -1.37 & 0 & -1.28 & 6.12 & -1.78 & 0 & -1.62 \\
0 & 0 & -1.37 & 0 & -1.78 & 7.37 & 0 & -1.62 \\
0 & 0 & 0 & -1.62 & 0 & 0 & 5.87 & -1.28 \\
0 & 0 & 0 & 0 & -1.62 & 0 & -1.28 & 6.62 & -1.78 \\
0 & 0 & 0 & 0 & 0 & -1.62 & 0 & -1.78 & 7.87
\end{bmatrix}
\]

\[
v = \begin{bmatrix}
0.343750 \\
0.625000 \\
3.437500 \\
0.062500 \\
0.062500 \\
2.593750 \\
0.179688 \\
0.531250 \\
3.648438
\end{bmatrix}
\]

Figure 7.1: Sample Matrix for \( Au = v \)

Matrix formed from the finite difference solution of the self-adjoint elliptic PDE
\[
\frac{\partial}{\partial x} (x + 1) \frac{\partial u}{\partial x} + \frac{\partial}{\partial y} (y^2 + 1) \frac{\partial u}{\partial y} - u = 1 \text{ on the unit square with Dirichlet Boundary conditions:}
\]
\[
u(0, y) = y, \ u(1, y) = y^2, \ u(x, 0) = 0, \ u(x, 1) = 1.
\]

The normal distribution function is chosen as the conditional distribution of erroring values. If we center the mean of the distribution at the actual correct result of the calculation, then the

\[\text{†Note that due to truncation of the values shown these values do not correspond exactly with the matrix shown.} \]
normal with mean $\mu$ and standard deviation $\sigma$ is an attractive choice since the erroring values tend to group near the correct value. While an overly pessimistic assumption, it does display the effectiveness of the generated constraint predicate. Figure 7.2 depicts the feasibility event $F$ and its relation to the conditional distribution of erroring values. $\mu = .562667$ is equal to the norm of the final solution vector $|u^{(k)}|/Q$. $\sigma = .65$ is assigned such that more than half of the error probability is undetectable by the feasibility conditions.

![Figure 7.2: Feasibility Event and Conditional Distribution of Errors](image)

Let $E$ be the event that an error has occurred. Figure 7.3 depicts the error coverage as a function of the iteration step $k$ for three different levels of error coverage. The error coverage of the feasibility component associated with Theorem 5.3 is constant since the constraints are stationary. The progress component rapidly rises to it’s limiting value of approximately 25% additional coverage. This is because the average solution makes the greatest reduction of error in the
early steps of the solution. The progress components associated with Theorem 6.3 and Corollary 6.1 add only a few more percent to the error coverage to bring the total error coverage to approximately 75%.

The choice of the normal as the distribution of errors also limits the error coverage using these predicates. Any symmetric distribution centered at the final result will necessarily result in an error coverage measurement somewhat less that total coverage since the tests developed (except for the progress component based on Corollary 6.1) are primarily one-sided.

The test associated with Theorem 6.5 has not yet been used in the constraint predicate. 6.5 affords total test coverage if certain additional restrictions are imposed. However, the fault latency rate is no longer one.

**VIII RUN TIME PERFORMANCE**

In matrix iterative solutions, many points will be assigned to a single processor. All the constraint predicates developed in this paper are applicable, but they only need to be run at those points falling at the boundary of the data partitioning. Let the total number of points on the grid \( Q \) be greater than the number of processors \( N \). Without loss of generality assume that \( N \) divides \( Q \) into a perfect square. Then \( Q/N = q \) is the number of points per processor. Continuing the assumption of coordinate grid indexed points, Figure 8.1 depicts the mapping and error checking for \( q = 25 \). The grids marked by dashed lines indicate points that are not checked by the constraint predicate as they are internal to that processor.
The unreliable algorithm, for a sub-grid of $q$ elements per processor with an average of $n-1$ neighbors, is easily seen to have a computational complexity per iteration given by,

$$C_{pNR} = nq + 2S_L$$

Each subclass of constraint predicate contributes the following run time overhead per iteration. From progress we have:

$$2 + n\sqrt{q}$$

From feasibility we have:

$$n\sqrt{q}$$

and from consistency we have:
The run time of the reliable algorithm is given by
\[ C_{pR} = nq + 2n\sqrt{q} + 2S_L + 3 \]

The ratio of the computation times of the reliable algorithm to the unreliable algorithm is:
\[ \frac{C_{pR}}{C_{pNR}} = \left[ \frac{nq + 2n\sqrt{q} + 2S_L + 3}{nq + 2S_L} \right] \]

Let \( r \) represent the computation to communication ratio. Then \( r = \frac{nq}{2S_L} \). Since, the other terms dominate \( \sqrt{q} \), then we that the ratio is approximately the following:
\[ \frac{C_{pR}}{C_{pNR}} = \frac{O(2S_L(r + 1))}{O(2S_L(r + 1))} = 1 \]

From this analysis, we can conclude that the addition of the constraint predicate to the unreliable algorithm does not significantly slow down run-time execution.

**VI. SUMMARY**

This paper has shown how to translate a verification proof into a fault-tolerant algorithm using the model of a matrix relaxation algorithm. It is easy to see that the progress and feasibility constraints of [McNi88] is directly derivable from the assertions of the verification proof, while the consistency constraint is derived from the need to ensure that each processor has a consistent view of the auxiliary variables used in the assertions of the verification proof.

Future research will further examine the relationship in the tradeoffs between minimizing fault latency and communication overhead. Up to this point, only synchronous programs have been examined. Future research will also examine asynchronous programs.
REFERENCES


1981, pp 118-120.


