FAULT-TOLERANT CONCURRENT BRANCH AND BOUND ALGORITHM
DERIVED FROM PROGRAM VERIFICATION †

Hanan Lutfiyya, Aggie Sun and Bruce McMillin

CSC 92-002

January 12, 1992

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

Abstract

The process of showing that a program satisfies some particular properties with respect to its specification is called program verification. Axiomatic semantics is a verification method that makes assertions describing properties about the states of the program. There exists a transformation from the assertions of the verification proof of a program to executable assertions. These executable assertions may be embedded in the program to create a fault-tolerant 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 distributed branch and bound as a model problem.

Keywords: Executable Assertions, Formal Methods, Branch & Bound, Concurrent Program Verification, Fault Tolerance.

† 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.
I. INTRODUCTION

The application-oriented fault tolerance paradigm [McNi88] is a promising approach to providing fault tolerance for the distributed 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 [YaCh75] 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 the 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 distributed environment is more complex than for the sequential environment. Since state information may only be communicated by message passing, the scope of the test that may be performed by an executable assertion is limited to testing received messages about a testing processor’s local state.

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 specification of the program to solve the problem. Thus, the executable assertions are closely tied to the individual components of the life cycle.

Working from this idea has led to a largely manual task of identification of appropriate assertions based on the three basis metrics of progress, feasibility, and consistency that formulate the constraint predicate paradigm. Progress is required to be made at each testable step of the solution. Progress means that the state of the solution advances to the goal or final solution of the problem. Each testable step of the solution is a message interchange, sub-message interchange, or multi-message interchange dependent on the type of solution. If this progress is not made, then faulty behavior may indefinitely postpone the solution. Each testable result must remain within the defined solution space of the problem. This is feasibility. The feasibility constraints are often immediately apparent from the problem studied. Problems in physics and engineering, such as equilibrium problems, eigenvalue problems, and to a lesser extent propagation problems, contain feasibility constraints as boundary conditions. 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 [LaSP82], 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.

The concept of a constraint predicate has been applied to distributed 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 and bound tree search in [SuMc91]. 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. Work focusing on the generation of assertions can be found in [LuMc91a, LuMc91b]. 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 [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. 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 a transformation similar to that outlined in [LuMc91c] from global auxiliary variables to the communication sequence-based system of [Soun84] to achieve consistency.

We illustrate the technique of transforming a verification proof outline of an algorithm to a fault-tolerant algorithm with branch and bound as applied to the N Puzzle Problem[Quin88]. Branch and bound is an effective technique to search for an optimal solution in an exponential search space. We chose to apply a branch and bound algorithm to the N puzzle problem because of the relative simplicity of expressing the solution of the N puzzle problem using branch and bound. We are not focusing on developing a more efficient parallel version of the branch and bound, but on the transformation process. This differs from other work concerned with the transformation process in that the message passing here is asynchronous.

The remainder of the paper is organized as follows: Section II discusses program verification and the application-oriented fault tolerance paradigm. Section III discusses the N puzzle problem using branch and bound. Section IV discusses a verification proof. Section V presents the main result of the paper, a transformation from the verification proof outline into a fault-tolerant program.

II. PROGRAM VERIFICATION AND APPLICATION-ORIENTED FAULT TOLERANCE

The axiomatic approach to program verification is based on 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. 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[Hoar69].

This paper uses the axiomatic proof system found in [LeGr81], which is used for Hoare’s model of concurrent programming, Communicating Sequential Processes(CSP)[Hoar78].

The proof system in [LeGr81] starts by proving appropriate properties about the individual processes in isolation. The next step is to use a rule of parallel composition to prove properties of the complete program. Proofs of the appropriate properties of individual processes require assumptions to be made about the effect of the communication commands. A “satisfaction proof” is then used to show that these assumptions are “legitimate”.

The proof system in [LeGr81] makes use of global auxiliary variables. Auxiliary variables 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. 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 requires a proof of “non-interference”.

Other axiomatic proof systems are 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 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.

The proof systems that have been discussed up to this point are designed for synchronous programming primitives. Our work uses an extension of work discussed in [ScSc84]. The work of [ScSc84] describes how to extend the notion of a “satisfaction proof” and “non-interference proof” for asynchronous message-passing primitives. The extension is based on introducing for each pair of processors i and j, two auxiliary variables $\sigma_{ij}$, $\rho_{ij}$, where $\sigma_{ij}$ is the set of all messages sent from process i to process j and $\rho_{ij}$ is the set of all messages j actually receives from i. This extension involves assuming that that actual sending and receipt of a message implies that $\sigma_{ij}$ and $\rho_{ij}$ are immediately updated. It is also assumed that $\rho_{ij} \subseteq \sigma_{ij}$ is invariantly true throughout program execution.

As will be shown in this paper, each of the 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. Testing is done through executable assertions where 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 arbitrary 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 component. Mathematically, this can be described as follows:

**Definition 2.1:** 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 2.2:** 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:

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

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

**Definition 2.3:** For a non-faulty process i, if exists any two tuples \( t_1, t_2 \in h_i \) such that

\[
\begin{align*}
  t_1 &= [j, (\text{Var}, \text{Val}_1), T, P_1] \\
  t_2 &= [j, (\text{Var}, \text{Val}_2), T, P_2]
\end{align*}
\]

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

In this paper, path inconsistent and path consistent will be used interchangeably with inconsistent and consistent, respectively.

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 2.3 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.

The other advantage of the consistency condition is based on the concept that if there are actually two tuples in \( h_i \) that satisfy the precondition, but \( \text{Val}_1 \neq \text{Val}_2 \), then a fault can be detected. This is useful for those cases in which an executable assertion may receive input values that are incorrect, but in which the executable assertions based on feasibility and progress can not catch. This occurs in problems where bounds are involved. In this case, the consistency condition is used as a means of strengthening these executable assertions.

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.

**III. BRANCH & BOUND ALGORITHMS**

Branch and bound algorithms have been used in the past to search optimal solutions for many well-known problems such as the Traveling Salesman and the N Puzzle Problem. These problems typically correspond to trees or graphs with exponential search space. There exists many search strategies that can find the optimal solution such as breadth-first or depth-first search, but the Branch and Bound approach uses a best-first search strategy, in that a function is used for estimating the node that is most promising. When a solution is obtained, the value of the function acts as an upper/lower bound to the problem where all other intermediary bounds can be pruned if it exceeds this bound. The algorithm described in this paper is the N Puzzle Problem where N+1 is a perfect square. The description of the problem is described
Most people are familiar with the N Puzzle Problem. The game begins with a given board configuration where the tiles are out of order. The objective is to find a solution that takes the minimal moves to go from the initial board configuration to a known final configuration. Figure 3.1 shows this.

![Figure 3.1: Objective of the N Puzzle Problem (N = 8)](image)

The initial board configuration for the N Puzzle Problem consists of N+1 tile positions with N tiles distinctly numbered ranging from 1 to N and one blank space denoted by a zero.

![Figure 3.2: An abstraction of the N Puzzle Problem (N = 8).](image)
The number of ways of reaching the objective is described best by means of a tree structure. The initial configuration corresponds to the root of a search tree (see Figure 3.2), where its children are the result of moving an adjacent tile into the blank position. A move which is legal is defined below.

**Definition 3.1:** A legal move is described by swapping the blank tile with a tile to its left, right, top or bottom. For each configuration, the following moves are legal if the conditions hold. There exists at most 4 possible moves per configuration. If \( b_{u,v} \) is the position of the blank tile then at least two of the following conditions will hold.

\[
\begin{align*}
  m_0 &: b_{u,v} \leftrightarrow b_{u+1,v} \quad \text{if } u \neq \sqrt{N+1} \\
  m_1 &: b_{u,v} \leftrightarrow b_{u-1,v} \quad \text{if } u \neq 1 \\
  m_2 &: b_{u,v} \leftrightarrow b_{u,v+1} \quad \text{if } v \neq \sqrt{N+1} \\
  m_3 &: b_{u,v} \leftrightarrow b_{u,v-1} \quad \text{if } v \neq 1
\end{align*}
\]

We will let \( M \) be the set \( \{ m_0, m_1, m_2, m_3 \} \).

Each node in the tree is referred to as a state, \( s_i \), of the puzzle where \( i \) is a unique integer. We will assume that \( s_0 \) denotes the state corresponding to the initial configuration. The state \( s_i \) will be represented by the path taken from the initial configuration, \( s_0 \), to the node represented by \( s_i \). A path is the sequence of moves from \( s_0 \) to the node \( s_i \). This can be formally defined as follows.

**Definition 3.2:** Let \( s_i \) of the N Puzzle Problem, be represented by the path that exists when a set of moves from the initial configuration, \( s_0 \), evolves into the node to be denoted by \( s_i \), where the number of moves taken is \( k \).

\[
s_i = ( p_{i0}, p_{i1}, p_{i2}, \ldots, p_{ik-1} ), \text{where } p_{ij} \text{ is a move of type } m, \text{ where } m \in M.
\]

**Definition 3.3:** A node denoted by \( s_i \) is a reachable configuration from a node denoted by \( s_j \) if \( s_j \) is a path prefix of \( s_i \).

If \( s_i \) is a reachable configuration from \( s_j \) then \( s_i \) and \( s_j \) are on the same path in the search tree.

**Definition 3.4:** Let the search space for the N Puzzle problem be described as follows:

\[
S_I = \{ s_i \mid s_i \text{ is a reachable configuration of } s_0 \}
\]

**Definition 3.5:** If \( s_i \in S_I \) then \( s_i \) is a solution if \( s_i \) corresponds to a node which denotes the final configuration.

**Definition 3.6:** If a solution exists, then the solution space, \( A_I \), is the set containing all solutions for the N Puzzle Problem.
\[ A_I = \{ s_i | s_i \in S_I \land s_i \text{ is a solution} \} \]

To search an entire tree for the optimal solution is exponential, therefore an optimization function is used to reduce the search space. This optimization function is applied to each path in the tree until the path containing the final configuration of the puzzle is found. The value of this function acts as a bound for that particular path down the tree and aids in selecting the next path to expand.

**Definition 3.7:** The optimization function, \( f \), determines the cost for a particular state, \( s_i \), at level \( k \).

\[
f(s_i) = m_d_i + k
\]

where \( m_d_i \) is the manhattan distance of the configuration denoted by \( s_i \).

The optimization function for the N Puzzle Problem is defined as the sum of the manhattan distance of each tile plus the height of the tree. This function represents the distance each tile is out of place and the number of moves taken thus far.

Theorem 3.1 states that if the branch and bound algorithm finds the minimal cost node as defined by the minimum cost function, which satisfies certain properties, then the minimal cost node is also the optimal node.

**Theorem 3.1:**[KoSt74] Let \( s \) denote any node representing a minimal cost node, according to the cost function \( f \), where \( f \) is monotonically nondecreasing. Then \( s \) is an optimal node in the search space.

**Theorem 3.2:**[SuMc91] The optimization function, \( f \) is a monotonic nondecreasing function as \( i \) increases.

By Definition 3.7 and Theorem 3.2, once a solution is found, the manhattan distance decreases to zero resulting in \( k \) for some \( f(s_i) \) evaluated at level \( k \). This value corresponds to the number of moves it took to solve the puzzle. The function value then acts as an upper bound to the problem such that all branches which possess a bound higher than the upper bound need not be considered. The nondecreasing nature of the function signifies that any path being considered with a higher bound, will take at least that many moves to solve the puzzle. Since a solution has already been found which can solve the puzzle in fewer moves, that path will not lead to a better solution. However, the remaining paths with lower bounds must continue expanding, generating new states until it reaches a solution or until it exceeds the current upper bound. When all the nodes of the tree have been explored, the solution having the lowest bound is the one with the minimum number of moves. The parallel algorithm described in this paper is based on this concept.

The parallel algorithm involves dividing the work in terms of subtrees and has many processors working on the problem simultaneously. Two approaches are described in this section.
The approach commonly taken in implementing Branch and Bound is the "worker/controller" method. The initial board configuration of the problem, $s_0$, is given to a designated processor called the "controller" who distributes the work to all idle processors known as "workers". The controller manages the tasks which are to be completed and is responsible for assigning tasks to the workers. A task refers to transforming some $s_i$ to $s_j$ by a legal move of type, $m_k$, where $m_k \in M$. A worker, on the other hand, is oblivious to the other workers and does only the task which is assigned to him by the controller. When a worker completes his task, he reports back to the controller by sending the controller the new $s_j$'s each with a bound accordingly to the $f(s_j)$. The controller receives these new $s_j$'s from the workers and inserts them into a prioritized queue which is in increasing order by the value of the bound, $f(s_j)$. When the controller recognizes that a particular worker is idle, he assigns the configuration with the lowest bound from his queue to that worker. As soon as a solution is found, the controller modifies his queue, disregarding all configurations with higher bounds. This process continues until the controller no longer has any tasks left to distribute. The current solution, $s_{current}$ which has the lowest bound then contains the optimal moves for the puzzle.

Host:
Begin
  Distribute initial board configuration;
  Terminate workers;
End;

Worker:
Begin
  Wait for a task to work on;
  Loop
    While ( task > 0 )
      Work on expanding lowest costing paths;
      If (first solution found)
        If $f(s_{task}) > f(s_{current})$
          Discard task.
        If a solution $s_j$ is found
          Notify other workers of $s_j$;
        If (task > 1) and (worker$_i$ = idle)
          Distribute task to idle worker$_i$;
        If a solution is reported
          If $f(s_{recv}) < f(s_{current})$
            $s_{current} = s_{recv}$;
      End While;
      Wait for a task to work on;
      If a solution is reported
        If $f(s_{recv}) < f(s_{current})$
          $s_{current} = s_{recv}$;
      End Loop;
End.

Figure 3.3: Parallel Branch and Bound for N-puzzle
The advantage of this scheme is the clear structural organization of the tasks. The controller knows the status of each worker and keeps track of the best bound reported thus far. However, the controller has become the point of centralization. All messages from the workers are directed to the controller creating a bottleneck. In terms of fault tolerance, the idea of having a centralized control is discouraged because should the fault manifest itself within that particular processor appointed to be the controller, recovery would be impossible.

We use as a model algorithm that of [SuMc91]. The algorithm requires only workers to search the solution space. The initial task, \( s_0 \), is assigned a designated worker to work on. As time passes, more tasks are created. Each worker retains one task for himself and redistributes the rest to other idle workers. When a worker has completed his task, he notifies the other workers that he is available to accept tasks; otherwise, he continues working on the original tasks distributed to him. The only other communication that occurs among the workers is when a solution, \( s_i \), has been found. This solution is only one of many in the solution space, \( A_1 \), and may not be the best solution, but it allows some pruning to be done such that the number of tasks can be reduced. The worker who discovers the solution broadcasts to the other workers allowing them to update their local bounds. The algorithm terminates when all tasks have either completed or been discarded. Asynchronous communication and dynamic allocation of tasks are used in this method. The algorithm is described Figure 3.3.

The dynamic allocation of tasks avoids the use of a centralized critical processor. If a processor should fail, the task that he was working on is easily redistributed to another processor. The asynchronous nature of the message passing allows workers to work independently without interruptions until there is an idle worker or he himself is idle. The advantage is the decrease in message passing required. Unfortunately, this aspect also makes detecting faults more complicated because it becomes difficult where timing is concerned.

### IV. VERIFICATION OF THE N PUZZLE PROBLEM

The algorithm presented in Figure 4.1, is the algorithm presented in Figure 3.3 annotated with assertions and assignments to auxiliary variables. Not all assertions are listed, since, there is one assertion, labelled I (defined in Definition 4.1), that is invariant throughout execution, except within during communication. This assertion and why it is not invariant throughout execution is described after Figure 4.1.
{PreH}
Host:
Begin
\[ S, A = S_1, A_1; \]
\[ S', A' = \emptyset, \emptyset; \]
Distribute initial board configuration;
\[ <S_0 \cup S_1 \cup \cdots \cup S_{N-1} = S_0> \]
Terminate workers;
End;
<PostH>

{Prew}
Worker:
Begin
Wait for a task to work on;
Loop
While ( task > 0 )
Work on expanding lowest costing paths;
If first solution found
If \( f(s_{task}) > f(s_{current}) \)
Discard task.
\[ T_0 = \{ s_i | s_i \text{ is a reachable configuration of } s_{task}\} \]
\[ T_1 = \{ s_i | s_i \in T_0 \land s_i \text{is a solution state}\} \]
\[ S_i, S', S = S_i - T_0, S - T_0, S \cup T_1 \]
If a solution is found
Notify other workers;
Update \( SB_{ij} \) for each \( j \);
If ( task > 1 ) and ( worker \( i \) = IDLE)
\[ T_0 = \{ s_i | s_i \text{ is a reachable configuration of } s_{task}\} \]
\[ S_i = S_i - T_0 \]
Distribute task to idle worker \( i \);
If a solution is reported
If \( f(s_{recv}) < f(s_{current}) \)
\[ s_{current} = s_{recv}; \]
Update \( R_{ij} \), where \( j \) is the sending process;
End While;
Wait for a task to work on;
\[ T_0 = \{ s_i | s_i \text{ is a reachable configuration of } s_{task}\} \]
\[ S_i = S_i \cup T_0 \]
If a solution is reported
If \( f(s_{recv}) < f(s_{current}) \)
\[ s_{current} = s_{recv}; \]
Update \( R_{ij} \), where \( j \) is the sending process;
End Loop;
solution = \( s_{current} \)
End.
<Postw>

Figure 4.1: Verification proof outline of the N puzzle problem
The following auxiliary variables are used in the verification proof:

- $S_I$: This is the set of all nodes in the tree that represents the state space.
- $A_I$: This is the subset of $S_I$ which contains the nodes that are solution nodes.
- $S'$: This is the set of nodes that have been examined by the algorithm, either directly or by pruning.
- $A'$: This is the set of solution nodes that have been examined by the algorithm, either directly or by pruning.
- $S$: This is the set of nodes that have not been examined by the algorithm.
- $A$: This is the set of solution nodes that have not been examined by the algorithm.
- $S_i$: This is the set of nodes to be examined by process $i$.
- $SB_{ij}$: This is the set of solutions sent from $i$ to $j$.
- $RB_{ij}$: This is the set of solutions received by $i$ from $j$.
- $\text{solution}_i$: This is the value of $s_{\text{current}}$ at the termination of worker $i$.

The precondition to the host process assumes that there is a solution from the initial configuration. In other words, $\text{Pre}_H$ is as follows:

$$< A_I \neq \emptyset >$$

For each terminating component process labelled node, $s_{\text{node}}$ is the lowest cost solution in the search tree. At the termination of the program, we want each processor to have the same lowest bound. Therefore, the postcondition $\text{Post}_H$ is represented as follows:

$$< \text{solution}_0 = \text{solution}_1 = \cdots = \text{solution}_{N-1} \land \text{solution}_i \text{ is the optimal cost solution} >$$

The precondition, $\text{Pre}_i$, to each worker process $i$ is the assumption that the worker processes initially have no tasks to examine and no communication has taken place. This is represented by

$$< S_i = \emptyset \land SB_{ij} = \emptyset \land RB_{ij} = \emptyset \text{ for all } j, 0 \leq j \leq N - 1, j \neq i >$$

The postcondition, $\text{Post}_i$ of the worker process $i$, is that the local variable $s_{\text{current}}$ has the following property:

$$< s_{\text{current}} \text{ is the optimal cost solution} >$$

Since $S$ and $A$ are the sets of nodes and solutions, respectively, that have yet to be examined; then since, at the beginning of the program none of the nodes or solutions have been examined, $S$ and $A$ are initialized to $S_I$ and $A_I$ respectively. Similarly, since $S'$ and $A'$ are the set of examined nodes and solutions, respectively; then since, at the beginning of the program none of the nodes or solutions have been examined, $S'$ and $A'$ are initialized to $\emptyset$.

The distribution of the initial board configuration corresponds to assigning to each process $i$, a subset of the nodes in the tree that represents the state space. No two processes should examine the same nodes. This corresponds to partitioning the auxiliary variable $S_I$ into the disjoint sets $S_0, S_1, \cdots, S_{N-1}$, which also satisfy the following immediately after initial board distribution:
In other words, all the search space nodes are distributed among all the worker process nodes. Since, the details of the board distribution are not included in the pseudocode, nor will the verification details.

Each worker process \( i \) must only examine those nodes that are part of the state space. Process \( i \) is presumed to examine nodes it either (1) Generated through expansion of other nodes or (2) Received from other processes. The first case implies that each new generated task must be part of the search space. For the sake of brevity, the details are not shown here, but it involves showing that each newly generated task is a reachable configuration from \( s_0 \) and hence, by Definition 3.4 considered to be an element of the search space represented by \( S_I \). We can conclude that the following is always true:

\[
< s_{\text{task}} \in S_I >
\]

Case 2 requires showing that the assertion is true after communication takes place i.e. the satisfaction proof. The details are not described, but it is intuitively easy to see by remembering that it will receive nodes to be examined from other processes, \( j \). We know that \( \sigma_{ji} \subseteq S_I \) is true, since \( \sigma_{ji} \) is immediately updated as the node is migrated from process \( j \) to process \( i \) and we know that all nodes migrated are members of \( S_I \) (Note that the definition of solutions implies that all solutions are members of \( S_I \)). In Section II, it was noted that \( \rho_{ji} \subseteq \sigma_{ji} \) is invariantly true. We can immediately conclude that

\[
< \rho_{ji} \subseteq S_I >
\]

Since, process \( i \) must examine only those nodes that are part of the search space then the following must be invariantly true:

\[
< S_i \subseteq S_I >
\]  \hspace{1cm} (1)

There are three instances when \( S_i \) is updated:

(1) When a part of the search tree is pruned off.
(2) When process \( i \) receives a node for expansion.
(3) When process \( i \) migrates a node to another process \( j \) for expansion by process \( j \).

For case (1), \( S_i \) is changed by first determining the set of nodes associated with the subtree to be pruned off. If the root node of the subtree to be pruned off is \( s_i \), then the set of nodes associated with the subtree to be pruned off is \( T_0 = \{ s_j \mid s_j \text{ is a reachable configuration of } s_i \} \). This set of elements is deleted from \( S_i \), i.e. the following operation is done: \( S_i = S_i - T_0 \). It is, therefore, easy to see that \( S_i \subseteq S_I \) is still true.

Showing that cases (2) and (3) maintain the truth of \( S_i \subseteq S_I \) is part of the satisfaction proof. Intuitively, this requires showing that the communication maintains the truth of \( S_i \subseteq S_I \). For case (2), process \( i \) receives a new task from process \( j \). It was earlier shown that that \( \rho_{ji} \subseteq S_I \). For case (2), \( S_i \) is updated by first determining the set of nodes associated with the subtree in which the received node is the root. This is done by determining all the reachable configurations from the received nodes. This set of nodes is added to \( S_i \). Since the received node is a member of \( S_I \) then all the reachable nodes from the received node are also members of \( S_I \). Hence, the truth of \( S_i \subseteq S_I \) is maintained. A similar argument can be made for case (3).
It is assumed that each node of the state space to be examined is assigned to a process. This is represented as follows:

\[ S_0 \cup \cdots \cup S_{N-1} \cup \{ s \mid s \in \sigma_{ij} - \rho_{ij} \wedge s \text{ is a task, where } 0 \leq i, j \leq N-1, i \neq j \} = S > \]  

(2)

Task migration does not change the set of nodes to be examined as a whole. Instead, a task migration only changes the set of nodes to be examined by the migrating and receiving worker processes. These changes were discussed in the previous discussion. Because of the asynchronous nature of the algorithm, it is possible for process \( i \) to send a task to process \( j \), but \( j \) is not ready to immediately receive the task. Therefore, it is necessary to include the set

\[ \{ s \mid s \in \sigma_{ij} - \rho_{ij} \wedge s \text{ is a task, where } 0 \leq i, j \leq N-1, i \neq j \} \]

It is necessary to ensure that the algorithm only examines those nodes that are in the state space. It is also necessary to ensure that all tasks and solutions are examined before they are discarded. This can be ensured by having the following invariantly true:

\[ S' \cup S = S_I \wedge A' \cup A = A_I > > \]  

(3)

The truth of this can be seen by observing that (1) The assertion stated in (3) is true at the beginning of program execution of the worker processes and (2) \( S, S', A \) and \( A' \) are updated when a subtree is pruned from the search space. The updating is done by determining all the set of nodes associated with the pruned subtree by finding all the reachable nodes from the root node of the subtree to be pruned off and determining all the solution nodes in the pruned subtree. The set of all nodes associated with the pruned subtree is deleted from \( S \) and added to \( S' \). The solution nodes are deleted from \( A \) and added to \( A' \). These updates are done simultaneously. Since, the assertion is initially true, then from the updates, it is obvious that the assertion stated in (3) is always true.

In the program, in each process \( i \), \( s_{\text{current}} \) is the solution that is known by process \( i \) to have the lowest bound. \( s_{\text{current}} \) is changed as more information about other solutions becomes known from other processes. Therefore, the following assertion is invariantly true:

\[ < s_{\text{current}} = \min (a \mid a \in R_i, \text{ where } R_i = \bigcup R_{bij}) > \]  

(4)

We would like to also ensure that the set of solutions received by each process by the termination of the program is equivalent. The following assertions aid in this. It must be invariantly true except when a solution is being broadcast (only because the auxiliary variable update is done after the broadcast).

\[ < S_{bij} = R_{ji} \cup \{ x \mid x \in \sigma_{ij} - \rho_{ij} \wedge x \text{ is a solution} \} > \]  

(5)

\[ < S_{b10} = \cdots = S_{b1n-1} > \]  

(6)

The assertion stated in (5) states that the solutions that are sent from process \( i \) to process \( j \) are the same received by process \( j \) from process \( i \) except for those solutions that are in transit. The assertion stated in (6) is true because process \( i \) sends a solution to all processes.

Since, \( A' \) is the set of solution nodes known not to be a lower bound, then for each solution node in \( A' \) there is at least one solution node in the set of broadcast solutions that is of lesser cost. Mathematically, this can be described as follows:

\[ < \text{For all } x \in A', \text{ there is an } i \text{ and } y \text{ such that } y \in R_i \text{ and } f(y) \leq f(x) > \]  

(7)
**Definition 4.1:** Let the assertion I denote the conjunction of the logical expressions expressed in (1)-(7).

The verification proof shows that I is invariantly true throughout user execution except for the part expressed in (5), (6). However, this is immediately rectified after the communication through assignments to $S_{ij}$.

The inner loop invariant for process $i$ is the following:

$$<( (\text{task} > 0 \land S_i \neq \emptyset) \lor (\text{task} = 0 \land S_i = \emptyset) ) >$$

The outer loop invariant for worker process $i$ is the following:

$$<( (\text{worker}_i = \text{busy} \land S_i \neq \emptyset) \lor (\text{worker}_i = \text{idle} \land S_i = \emptyset \land \sigma_{ij} = \emptyset) ) >$$

It can be shown that the termination of the Host process implies that each worker processor knows the optimal cost node. First, it is shown that the termination of the program in Figure 3.3 implies that each processor received the same set of broadcast solutions.

**Lemma 4.1:** At the termination of the program in Figure 3.3 the following assertion is true:

$$< R_0 = \ldots = R_{N-1} >$$

**Proof:**

The outer loop invariant and termination of all worker processes implies that we have $\sigma_{ij} = \rho_{ij} = \emptyset$. Hence, for a process $i$ and any worker process $j$, we have that $SB_{ij} = RB_{ij}$. From (7), we can conclude that at the termination of the program that $RB_{0i} = \ldots = RB_{N-1i}$. This is true for any worker process $i$. Since, for any worker process $j$, $R_j = \bigcup RB_{ji}$, then we can conclude that $R_0 = \ldots = R_{N-1}$

Now, Lemma 4.1 and Theorem 3.1 can be used to show that the termination of the program in Figure 3.3 implies that each worker processor has found the optimal solution.

**Theorem 4.1:** The termination of the program in Figure 3.3 implies the postcondition, $\text{Post}_{H}$

**Proof:**

At the termination of the outer loop, we have that all processes are idle and hence, for all $i$, where $0 \leq i \leq N - 1$, we have that $S_i = \emptyset$. Since, the verification proof shows that $S_0 \cup S_1 \cdots \cup S_{N-1} = S$, then we can conclude that $S = \emptyset$. Since, $A_{\text{current}} \subseteq S$ and $A \cup A' = A_f$, then we can conclude that $A' = A_f$. From (4), (7) and Lemma 4.1 we can easily conclude that $s_{\text{current}}$ is the same in each process and is the minimal cost as defined by the optimization function defined in Definition 3.7. From Lemma 4.1 and Theorem 3.1, we know that this minimal cost is the optimal cost.

V. Translation

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 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. The resulting set of assertions after the deletion of those using only local variables will be embedded into the operational environment as executable assertions. These assertions will become the progress($\Phi_P$) and feasibility($\Phi_F$) constraints.

3. Communicate the variables used in the assertions of 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.

We chose to avoid working with assertions that only use local variables based on the following: auxiliary variables are generally used to record part of the history of communication. Assertions that use only local variables limit the operational fault-tolerant environment to having each node check only itself. Treating global variables affords us inter-node diagnosability.

The transformation can be done algorithmically[LuMc92]. But, this results in fault-tolerant programs in which the run-times are significantly increased for the non fault-tolerant version of the program. This tradeoff between speed and reliability is not justifiable for many applications. Therefore, it becomes necessary to examine heuristic means for reducing the runtime execution. The heuristics are based on choosing a subset of the assertions using global auxiliary variables based on the following criteria.

The first criterion is based on having processes do more than just 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 have a consistent view of the data.

The second criterion is based on the desire to minimize the 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 instrumented assertions. Then, if we have 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 expression:

$$\text{length}(H'_i) - \text{length}(H_i)$$

The desire to minimize extra communication steps implies that input values of executable assertions should be piggybacked along in the natural communication. Since, in most distributed systems, the
overhead is in the communication setup and not in the message length, it can then be seen why it is desirable to minimize extra communication steps.

In general, it is possible for the criteria used in choosing assertions to be in direct conflict. This then makes it necessary to use weaker conditions.

For some applications, including the branch and bound, it is difficult to translate some assertions for the following reason: There are auxiliary variables that are updated by different processes. No one process has a complete picture of such auxiliary variables. For example, in the verification proof in the previous section, the auxiliary variable $A'$ records the set of solutions that have been examined. Each process knows what it has contributed to this set, but does not have a complete picture of this set. This leads to the global snapshot problem, which would incur a significant amount of overhead of communication. This is in direct conflict with the desire to minimize extra communication steps.

Also, related to minimizing communication overhead is the issue of the frequency of executing the execution assertions based on the invariant. For example, should execution be done after each execution of the inner and outer loops? This would increase communication overhead, since not in all loop iterations does communication take place. The approach taken is to execute the executable assertions every time a path is communicated or a solution is broadcast.

VI. Deriving Progress and Feasibility

The results of this section can be summarized in Table 1. The executable assertions are executed after the receipt of a message and $s_i$ denotes a received task/solution. The rest of this section expands on the transformation.

Let us now examine the executable assertions that can be derived from the verification proof. First, we will examine executable assertions directly derived from the invariant. The first part to be examined is the following:

$$< S_i \subseteq S_I >$$

In the previous section, we observed that the nodes to be examined (that is what $S_i$ consists of) changes as the result of either pruning, receiving a new task or the migration of a task. As a result, there are two types of fault occurrences. If process $i$ is faulty then the wrong subtree may be pruned. If the pruned subtree does not contain the optimal solution then it does not matter. Later, an executable assertion will be discussed that can catch when the optimal solution was pruned off. The other type of fault occurs when a task was communicated wrong. Hence, process $i$ could either receive a task incorrectly communicated or a task that process $i$ migrates could be incorrectly migrated. Therefore, it becomes feasible for every process that receives a task to check to see whether that task represents a node in the search space. From Definition 3.2, a migrated task representing $s_i$ is represented as follows:

$$s_i = (p_{i0}, p_{i1}, p_{i2}, \ldots, p_{ik-1})$$

where $p_{ij}$ is a move of type $m$, where $m \in M$.

and communicated according to Definition 2.1 by

$$[j, s_i, T, P]$$

Therefore, a receiving process can check whether each move is legal (as defined by Definition 3.1) by first applying the move $p_{i0}$ to the initial configuration. Assuming that head($s_i$) extracts the first element in the
Transformation of Assertions

<table>
<thead>
<tr>
<th>Verification Assertion</th>
<th>Executable Assertion</th>
<th>Transformation</th>
</tr>
</thead>
</table>
| $\forall s_i \in S_1 \land S_0 \cup \cdots \cup S_{N-1} \cup \{s \mid s \in \sigma_{ij} - \rho_{ij} \land s \text{ is a task}, \text{where } 0 \leq i, j \leq N-1, i \neq j\} = S \land S' \cup S = S_1 \land A' \cup A = A_1 \land s_{current} = \min (a \mid a \in R_i, \text{where } R_i = \bigcup \cup R_{ij}) \land SB_{ij} = R_{ji} \cup \{x \mid x \in \sigma_{ij} - \rho_{ij} \land x \text{ is a solution} \} \land SB_{i0} = \cdots = SB_{iN-1} \land \forall x \in A', \text{there is an } i \text{ and } y \text{ such that } y \in R_i \text{ and } f(y) \leq f(x) >$
| $\Phi_{F_1}: s_i \in S_1$ | Directly from $S_i \subseteq S_1$ |
| $\Phi_{F_2}: s_i \in A_1$ (assuming that $s_i$ is a proposed solution) | Result of weakening the condition: $\forall s_i \in S_1 \land A' \cup A = A_1 \land s_{current} = \min (a \mid a \in R_i)$ |
| $\Phi_{F_3}: (R_{ij} \subseteq S_{ij})$ | Result of weakening the condition: $\forall x \in A', \text{there is an } i \text{ and } y \text{ such that } y \in R_i \text{ and } f(y) \leq f(x)$ |
| $\Phi_{F_4}: \text{cost}_0 = f(s_i)$, where $\text{cost}_0$ is the received cost of $s_i$ | Result of weakening the condition: $\forall x \in A', \text{there is an } i \text{ and } y \text{ such that } y \in R_i \text{ and } f(y) \leq f(x)$ |
| $\Phi_{F_5}: f(s_i) > f(s_{prev})$, where $s_{prev}$ is the previously proposed solution | Result of weakening the condition: $\forall x \in A', \text{there is an } i \text{ and } y \text{ such that } y \in R_i \text{ and } f(y) \leq f(x)$ |

Table 1: Summary of Executable Assertions

list that $s_i$ represents and that tail($s_i$) extracts everything but the first element from $s_i$, the pseudocode carrying out this check is as follows:

It is possible to have a task that is incorrectly communicated that still represents a node in the search space. If the node is the root of a subtree that does not contain the optimal solution, then it does not matter. Later, an executable assertion will be discussed that can catch when the node is the root of a subtree that does contain the optimal solution.

The following part of the invariant assertion

$$< S' \cup S = S_1 \land A' \cup A = A_1 >$$

poses implementation problems because of the use of auxiliary variables in which no one process has a complete knowledge of. Instead, a weaker condition is used. For each migrated node or solution, it is
If $p_0$ is applicable to $s_0$ then
   apply $p_0$ to $s_0$ and assign to $n$
else
   ERROR;

$s_i$ = tail($s_i$);
while $s_i$ is not empty
   if head($s_i$) is applicable to $n$
     apply head($s_i$) to $n$ and assign to $n$
     $s_i$ = tail($s_i$)
   else
     ERROR
end while;

Figure 6.1: $\Phi_{F_1}$: Pseudocode to check whether a sequence of moves is legal

possible to test whether that node belongs to the search space by using the pseudocode of Figure 6.1. It is also desirable to check whether a solution node is really a solution. Assume that $goal_{u,v}$ and $t_{u,v}$, where $u$ and $v$ range from 1 to $\sqrt{N} + 1$, represent the goal configuration and configuration of the received solution node, respectively. Then the above assertion induces the executable assertion of checking whether the received solution node is really a member of $A_1$. The following pseudocode can be used to implement this executable assertion:

For ($u=1; u<=\sqrt{N}+1; u++$)
   For ($v=1; v<=\sqrt{N}+1; v++$)
      If ($t_{u,v} \neq goal_{u,v}$) then ERROR;

Figure 6.2: $\Phi_{F_2}$: Check to see whether a proposed solution is really a solution.

The next part of the invariant to be examined is the following:

\[ < s_{current} = \min (a | a \in R_i) > \]

Whenever, $s_{current}$ is to be broadcast, the value of $R_i$ is also broadcast, as $[j, R_i, T, P]$. $s_{current}$, on arrival at all other nodes, can be tested to see if it is the minimal value in $R_i$. This induces the following:

If $s_{current} \neq \min (a | a \in R_i)$ then ERROR;

Figure 6.3: $\Phi_{F_3}$

The following assertion

\[ < SB_{ij} = R_{ji} \cup \{x | x \in \sigma_{ij} - \rho_{ij} \land x \text{ is a solution} \} > \]

is difficult to translate in a runtime environment into an executable assertion because of the difficulty of determining the status of $\sigma_{ij}$ and $\rho_{ij}$. Instead, the following weaker condition is transformed into an
executable assertion:
\[ R_{ji} \subseteq SB_{ij} \]
If \( R_{ji} \) and \( SB_{ij} \) are piggybacked with solutions as \([k, R_{ji}, T, P]\) and \([k, SB_{ij}, T, P]\), respectively, then on receipt of the solutions, the following pseudocode is executed:

\[
\text{If } \neg (R_{ji} \subseteq S_{ij}) \text{ then ERROR;}
\]

**Figure 6.4: \( \Phi_{F_4} \)**

The following assertion
\[
\langle \text{For all } x \in A', \text{ there is an } i \text{ and } y \text{ such that } y \in R_i \text{ and } f(y) \leq f(x) >
\]
is carried out using a weaker condition. The above implies that for a process \( i \), the successive values of \( s_{\text{current}} \) are monotonically nondecreasing. Since, the goal is to find the solution with the least moves, each solution received should exhibit a better way of solving the puzzle than the previous. If previously broadcast solutions for each worker process (denoted by \( s_{\text{prev}} \) in the pseudocode) is saved and if \( s_i \), as communicated by \([j, s_i, T, P]\), denotes the received solution then the following pseudocode implements the above condition:

\[
\Phi_p: \text{IF } s_i \text{ is a solution then}
\]
\[
\text{IF } s_i > s_{\text{prev}} \text{ then ERROR;}
\]

**Figure 6.5: \( \Phi_p \)**

It is also desirable to ensure that costs are being computed correctly. If \( \text{cost}_0 \) is the communicated cost of a migrated node or broadcast solution and that \( s_i \) is the migrated task or broadcast solution then the following pseudocode can be used to ensure that costs are being computed correctly.

\[
\text{Let } \text{cost}_1 = f(s_i);
\]
\[
\text{If } \text{cost}_0 \neq \text{cost}_1 \text{ then ERROR;}
\]

**Figure 6.6: \( \Phi_{F_5} \)**

The executable assertions denoted by \( \Phi_{F_1}, \Phi_{F_2}, \Phi_{F_3}, \Phi_{F_4}, \) and \( \Phi_{F_5} \) measure the validity of the information. Hence, these executable assertions fall into the feasibility class of executable assertions. \( \Phi_p \) is based on the goal of finding the solution with the least moves. Thus each solution received should exhibit a better way of solving the puzzle than the previous. Therefore, \( \Phi_p \) is an example of a progress constraint. Note that \( \Phi_{F_1} \) and \( \Phi_{F_5} \) are applied to migrated tasks and solutions, while \( \Phi_{F_2}, \Phi_{F_3}, \) and \( \Phi_{F_4} \)
and $\Phi_P$ are applied to broadcast solutions. In section II, we discussed the importance of consistency or ensuring that if two non-faulty processors execute the same executable assertion, then the result is the same. The next section will discuss consistency for the executable assertions discussed up to this point.

Earlier we discussed cases in that the optimal solution may be pruned off or that the subtree that the optimal solution is in is not communicated properly. This would imply that the postcondition of the program, $Post_{HT}$, is false. This is the result of a processor never correctly broadcasting the optimal solution to the other workers. This case is coined the “silent worker” scenario. One way of testing the postcondition would be for each process to broadcast its perception the best solution to all other processes. One way of ensuring the consistency condition would be to use the Byzantine General’s algorithm[LaSP82]. This is clearly infeasible. Instead a verification round (or stage) is used to check the validity of the solution. This is handled efficiently by the following procedure, that is called the verification stage[SuMc91]:

$$\begin{align*}
\text{if } \forall[i, s_{current}, T, P] \neq s_{current} \text{ then} \\
\text{ERROR!}
\end{align*}$$

**Figure 6.7:** $\Phi_{F_6}$: Testing the postcondition

The verification stage consists of resolving the puzzle by redistributing the initial states to different workers and restricting the workers to communicate within disjoint sets of workers. This is necessary in order to keep a silent worker from retaining the optimal solution again. However, a solution has already been found that has a bound lower than most paths in the tree. Therefore, many paths need not be considered in the verification round if the bound exceeds the previous optimal bound. There may be more than one "silent worker", thus the number of verification stages necessary depends on the upper bound on the number of "silent workers" allowed. Each verification round allows for each process to recompute the solution in different ways. Thus, if there is a discrepancy in the values received in the different rounds then there is an error. For full details of the fault-tolerant algorithm using verification rounds see [SuMc91].

This section describes the development of the assertions necessary to cover the progress and feasibility constraints for Application-Oriented Fault Tolerance. Each assertions was developed based on the definitions of the problem and the behavior they exhibit as defined by the verification proof. The algorithm annotated with the executable assertions derived in this section appears in Figure 6.8.

### VII. Consistency from Natural Redundancy

Path consistency may either be explicit or implicit. In other words, a fault-tolerant program may have to explicitly add code to implement path consistency. This can be done in many ways. On the otherhand, there are classes of problems that have the property of natural redundancy in the problem variables. This implies that there are types of errors that if they occur in state $i$, then eventually, at some state $j$ (where $j > i$), we have that state $j$ satisfies the properties as defined by the intermediate assertions of a verification proof, despite the error that had occurred in stage $i$. If a program variable is naturally redundant then this means that this program variable can be constructed from other variables. The following is a formal method of describing redundancy.
Worker: Begin

- Wait for a task to work on;
- Call \( \Phi_{F_1} \) (task);
- Call \( \Phi_{F_2} \) (task);

Loop

- While (task > 0)
  - Work on expanding lowest costing paths;
  - If (first solution found)
    - If \( f(s_{\text{task}}) > f(s_{\text{current}}) \)
      - Discard task;
    - If a solution \( s_i \) is found
      - Notify other workers of \( s_i \);
  - If (task > 1)
    - Distribute task to idle workers;
    - If a solution is reported
      - If \( f(s_{\text{recv}}) < f(s_{\text{current}}) \)
        - Call \( \Phi_{F_2} \) (solution);
        - Call \( \Phi_{F_3} \) (solution);
        - Call \( \Phi_{F_4} \) (solution);
        - Call \( \Phi_P \) (solution);
        - \( s_{\text{current}} = s_{\text{recv}} \);

End While;

- Wait for a task to work on;
- Call \( \Phi_{F_1} \) (task);
- Call \( \Phi_{F_2} \) (task);

If a solution is reported
- if \( f(s_{\text{recv}}) < f(s_{\text{current}}) \)
  - Call \( \Phi_{F_2} \) (solution);
  - Call \( \Phi_{F_3} \) (solution);
  - Call \( \Phi_{F_4} \) (solution);
  - Call \( \Phi_P \) (solution);
  - \( s_{\text{current}} = s_{\text{recv}} \);

End Loop;

End.

Call \( \Phi_{F_6} \)

Figure 6.8: Fault-Tolerant Program

In this section the concept of a naturally redundant algorithm is generally defined and then it will be seen why the redundancy implies consistency.

**Definition 7.1** [LaMG91]: If a given algorithm \( A \) maps an input vector \( X = (x_0, x_1, \ldots, x_{N-1}) \) to an output vector \( Y = (y_0, y_1, \ldots, y_{N-1}) \) and the redundancy relation \( \forall y_i, y_i \in Y, \exists F_i | y_i = F_i(Y - \{y_i\}) \) holds, then \( A \) is called a Naturally Redundant Algorithm. Each \( x_i(y_i) \) may be either a single component of the input(output) or a subvector of components.

From this definition we can see that a naturally redundant algorithm running on a processor architecture \( P \) has at least the potential to restore the correct value of any single erroneous component \( y_i \) in its
output vector. This will be the case when each $F_i$ is a function of every $y_j, j \neq i$. If each $F_i$ is a function of only a subset of the components of $Y - \{y_i\}$ then the algorithm would potentially be able to recover more than one erroneous $y_i$.

In the parallel execution of many applications processors communicate their intermediate calculation values to other processors as the computation proceeds. In such cases, the erroneous intermediate calculations of a faulty processor can corrupt subsequent computations of other processors. It is thus desirable, that the correct intermediate calculations could be recovered before they are communicated to other processors. This motivates the definition of algorithms that can be divided in phases that are themselves naturally redundant.

**Definition 7.2 [LaMG91]:** An algorithm $A$ is called phase-wise naturally redundant algorithm if: (a) Algorithm $A$ can be divided in phases such that the output vector of one phase is the input vector for the following phase; (b) The output vector of each phase satisfies the redundancy relation.

An algorithm may be naturally redundant in a strict, loose or specification-wise. The value of a component of a phase output vector calculated by the redundancy relation is strictly correct if it is exactly equal to the value (correctly) calculated by the algorithm. It is loosely correct if it is not equal to the value calculated by the algorithm, but its utilization in subsequent calculations will still lead to the expected results (those that would be achieved if only strictly correct values were used). Finally, it is specification-wise correct if it is not equal to the value computed by the algorithm and its further utilization does not lead to the expected results, but to results that satisfy system specification.

The natural redundancy nature of program variables provides recovery from errors in future states of program execution. Natural redundancy allows for a forward recovery approach [Mili85], since there is no need of backtracking the computation to restore a correct value of an erroneous output vector component.

Earlier we discussed that consistency provides that any two processors executing the same executable assertion reach the same conclusion and for strengthening executable assertions based on progress and feasibility. The executable assertions require input values to be communicated. If we do not implement the consistency condition as described, earlier in this section, then it is possible for the cases described in Figures 7.1, 7.2(a) to occur. In these cases, the bounds received by processor $A$ is incorrect. It is possible that the progress and feasibility executable assertions will not catch the type of error illustrated in these figures. However, it can be shown that the N Puzzle algorithm is phase-wise naturally redundant in the loose sense with respect to the broadcast of the solutions. This implies that if a solution is incorrectly broadcast in state $i$ of the program execution, then the program execution will correct itself by a later state $j$.

For the N puzzle algorithm annotated with the executable assertions derived in Section VI, a phase is between broadcast solutions. It is assumed that the input vector is $X = (s_{\text{current}}, \ldots, s_{\text{current}})$ and that the output vector $Y = (s_{\text{current}}, \ldots, s_{\text{current}})$, where $X$ and $Y$ each have $N$-1 components. This is to imply that if process $i$ broadcasts its lowest solution, then it is broadcasting the same solution to all other processes. In other words, the broadcast represents the mapping and the mapping is an identity relationship.

**Theorem 7.1:** The N Puzzle algorithm of Section III annotated with the executable assertions developed in the Section VI and displayed in Figure 6.8 is a phase-wise naturally redundant algorithm in
To show that the algorithm is a phase-wise naturally redundant algorithm in a loose sense requires showing that if a solution is incorrectly broadcast its utilization in subsequent calculations will still lead to the expected results. There are several cases to consider.

**Case 1:** A value of $s_{\text{current}}$ (where $s_{\text{current}}$ is a broadcast solution) is received by worker A that is higher than the value sent by worker B when all other workers receive correct lower values. Worker A discards all solutions with bounds higher than $s_{\text{current}}$ and continues expanding the rest of the subtree. As soon as all the workers have completed their assigned paths, each compares his own view of the optimal solution with those received by the other workers. The solution with the lowest cost is the optimal solution to the problem. Since worker A’s solution is higher than the rest, A’s solution will never be considered.

![Diagram](image)

**Figure 7.1:** Bound received by worker A is greater than all the others.

**Case 2:** Now suppose the cost received by worker A is lower than all the rest. This must be a correct solution or it will be flagged by the progress or feasibility constraints. All other workers will continue working based on the best solution known to him. The effects of this scenario will only slow the process down as a whole, but will not cause any candidate solutions to be disregarded. The algorithm self-corrects itself when:

1. A new solution is broadcast whose value is less than A’s current bound. Other workers then update their knowledge of the current best bound as well.

![Diagram](image)

**Figure 7.2** a. The bound received by worker A is lower than all other workers. b. Worker A self corrects itself when a lower bound is discovered.
2: Worker A completes his job before a better solution is found, in which case, his result will be the optimal solution to the problem.

<table>
<thead>
<tr>
<th>Workers</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>...</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>.</td>
</tr>
</tbody>
</table>

Figure 7.3: Bound received by worker A is lower than all other workers at the completion of the first round.

Cases 1 and 2 show that if the current best perceived by one worker is correct but differs from the global view of the rest of the workers, then it will either self-correct itself when a new solution is broadcast or wait until the completion of the first round to check with the others. Hence, consistency is inherent in the algorithm and no additional constraints are necessary at this point.

It can also be shown that errors in the transmission of the other variables (the ones to be used for the executable assertion) implies that error in the transmission of a solution.

We have shown that the Fault-Tolerant program of Figure 6.8 is naturally redundant, hence, there is no need for an explicit consistency constraint predicate.

VIII. SUMMARY

Performance is a key issue when evaluating a fault-tolerant algorithm. It is obvious that some cost of overhead time cannot be avoided, but a good fault-tolerant algorithm will provide minimal overhead. Experimentation on the parallel fault-tolerant algorithm shows that the amount of overhead was mainly caused by the verification stage, as expected. However, the fault-tolerant algorithm is efficient. Experimentation results are presented in [SuMc91].

This paper has shown how to translate a verification proof into a fault-tolerant algorithm using the model of a branch and bound problem. This problem showed how executable assertions are related to the intermediate assertions of a verification proof. This problem also showed how the consistency condition can be relaxed when there is natural redundancy in the program variables.

Future research will further examine the relationship in the tradeoffs between minimizing fault latency and communication overhead.

REFERENCES
