AN APPLICATION-ORIENTED APPROACH TO DISTRIBUTED ERROR-DETECTING BRANCH & BOUND†

Aggie Y. Sun, Hanan Lutfiyya, and Bruce McMillin

CSC-92-24

December 18, 1992

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 Number MSS-9216479, and, in part, from the Air Force Office of Scientific Research under contract number F49620-92-J-0546.

Hanan Lutfiyya is with the Department of Computer Science at the University of Western Ontario, London, ONTARIO N6A 5B7 (hanan@csd.uwo.ca). The work was performed when Dr. Lutfiyya was at the University of Missouri-Rolla.

Aggie Sun (aggies@cs.umr.edu) and Bruce McMillin (ff@cs.umr.edu) are with the Department of Computer Science at the University of Missouri-Rolla, Rolla, MO 65401.
ABSTRACT

An important aspect which is often overlooked in software design of distributed environments is that of fault tolerance. Many methodologies in the past have attempted to provide fault tolerance efficiently, but have never been successful at eliminating explicit time and space redundancy. One approach is the Application-Oriented Fault Tolerance Paradigm, which provides fault tolerance by examining the behavior and properties of the application and deriving executable assertions for the detection of faults. Previous work has demonstrated the feasibility of the application-oriented fault tolerance paradigm for various applications. However, the executable assertions were guided by the natural constraints of the problem. This work uses Changeling to apply concurrent programming axiomatic proof systems to formally generate executable assertions in a distributed environment, by showing a transformation of the assertions derived from the verification proof of a program into executable assertions. These executable assertions are then embedded into the program to create a fault-tolerant program. The model used to demonstrate this approach is the class of Branch and Bound problems. The resulting error-detecting algorithm is implemented in parallel on a distributed memory machine. Experimental performance results and analytical bounds on the fault tolerance are reported for the Traveling Salesman application.

KEYWORDS: Distributed Algorithms, Fault Tolerance, Branch and Bound, Error Detection, Application-Oriented Techniques, Naturally Redundant Algorithms, Loose Synchrony, Performance Validation
I. INTRODUCTION

The requirements of increasingly complex and sophisticated concurrent systems demand fault tolerance techniques to ensure dependability. Dependability states that a system performs as expected. A system may deviate from its expected behavior when a fault occurs. To be tolerant of faults means that a fault is allowed to appear, but when it does, the system is able to detect it and effect a recovery such that the system does not fail. The first step in this process is to detect the presence of an error in a component of the system.

Previous strategies of providing fault tolerance require redundant hardware or software to improve overall system reliability [HoSL78, Wens78, ChAv83]. The use of redundant hardware can detect and recover from hardware component failures, but cannot cope with faults that might occur due to software. Classic methods for software fault tolerance, such as N-version programming [ChAv83] are limited as discussed in [AmKB89]. Additionally, these strategies incur a heavy hardware/software overhead penalty due to their explicit replication of resources. Thus, as systems grow in size and complexity, the idea of redundancy becomes less appealing.

Software implemented fault tolerance detects faults in the software as well as failures in the underlying hardware that manifest themselves as software errors. This usually takes the form of executable assertions (if-then-else statements) on the computational results instead of using explicit redundancy. Once an error has been detected by the executable assertion, fault location, reconfiguration, and recovery logically follows. Previous work in executable assertions include self-checking software by [YaCh75] and recovery blocks by [Rand75], both of which instrument the software with assertions on the program’s state, watchdog processor by [MaML83], which monitors intermediate data of a computation, and algorithm-based fault tolerance by [HuAb84], which imposes an additional structure on the data to detect errors. These methods define structure for fault tolerance, but do not, generally, give a methodology for instantiating the structure.

The work in the Application-Oriented Fault Tolerance paradigm, as proposed by [McNi88], provides a strategy for determining the executable assertions. There are three properties of algorithms exploited by Application-Oriented Fault Tolerance: feasibility, progress, and consistency. Based on these three properties of the algorithm’s specification, executable assertions are developed and incorporated into the algorithm to detect errors. The advantage of using this strategy is its applicability in a parallel and distributed environment. Since centralized processing is generally avoided in any fault-tolerant system, error detection should be distributive in nature, based on peer to peer evaluations. In a distributed parallel system, executable assertions make sense only at the communication boundaries between processors, since a processor cannot reliably check itself. Since the bulk of the overhead is essentially due to the number of messages passed, and the assertions themselves generate little overhead, the approach yields a low cost error-detecting algorithm.

This concept 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 [McNi89] and parallel sorting [McNi92]. What is needed, however, to make the concept viable, is a formal method for generating executable assertions. Formal methods for the generation of assertions, embodied by the Changeling strategy [LuSM92] is used in this paper. In particular, using Changeling, a proof outline from the verification of a concurrent program is used as a starting point to generate the fault-tolerant assertions.

Previous applications of Changeling [LuMc91b,LuSM92,LuSM92a] have dealt with problems having tight synchronization. This paper treats, by contrast, the application of Changeling to asynchronous, naturally redundant, algorithms, specifically, the class of Branch and Bound algorithms. The goal, however, is not to develop a new parallel Branch and Bound technique, but to show the application of Changeling and to validate that the assertions generated by Changeling are efficient. Additionally, the resulting fault-tolerant algorithm is interesting as there has been only one other work in building fault-tolerant Branch and Bound algorithms; [Vorn87] considered a fault-tolerant branch and bound using message absence as an acceptance test. This paper goes beyond simple message presence and considers message content as well.

The paper is organized as follows. A survey of program verification techniques and how Changeling employs them in the generation of fault-tolerant program is presented. The class of Branch and Bound algorithms and a dynamic load balancing formulation of the N puzzle model problem solution. Third, the development of the assertions through program verification and application of Changeling is shown. Fourth, an error-detecting algorithm for the more realistic Traveling Salesman application is proposed. Finally, a model is developed to measure the performance of the error-detecting algorithm.

II. CHANGELING

[Mili81] was the first to utilize formal methods to show fault tolerance through software specified executable assertions. Similarly, program verification provides a framework for generating assertions. 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. Proofs of theorems required for program verification are of the following form:

\[<P>S<Q>\]

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, upon termination of S, Q is true. P and Q are said to be the \textit{precondition} and \textit{postcondition} of S, respectively [Hoar69].

The usefulness of the assertions developed in the verification proof is extended by embedding the assertions into the resulting code for testing at run time. While this approach has been applied to the sequential programming environment, the distributed programming environment presents special challenges. In the distributed environment, not every postcondition, Q, from the
proof outline can be tested at run time; complete state knowledge is not available for generating executable assertions. Two problems are encountered in making the transition to the distributed environment:

1) Selection of Relevant State Information
2) Reliable Communication of State Information

Changeling represents the work addressing these two problems and consists of the following five distinct components:

- A shared memory verification system (GAA) based on Levin & Gries’ work
- The HAA proof system which mimics closely the distributed operational environment
- Formal conversion from GAA to HAA
- Formal translation of assertions in the HAA proof system to executable assertions
- Assessment of the fault-tolerant concurrent program

These five points are addressed in the remainder of this section. The axiomatic proof system found in [LeGr81], which is used for Hoare’s model of concurrent programming, Communicating Sequential Processes (CSP) [Hoar78] forms the basis for the verification system used in this paper.

### 2.1. GAA Proof System

A proof outline employing global auxiliary variables [LeGr81] as a proof aid is, perhaps, the easiest approach to reason about concurrent systems. In this system, processes in isolation are reasoned about. For a full concurrent proof of the appropriate properties of individual processes, assumptions are required to be made about the effects of process interleaving and auxiliary variable assignments on the validity of preconditions and postconditions on the communication commands. A “satisfaction proof” is then used to show that these assumptions are “legitimate”. Hence, a parallel inference rule is as follows:

\[
\frac{(\forall i: <P_i> S_i <Q_i>)}{<(\forall i: P_i)> \mathcal{L}_{i=1:n} A_i :: S_i <(\forall i: Q_i)>}
\]

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. The proof system in [LeGr81] makes use of global auxiliary variables (GAVs). Auxiliary variables can neither affect 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 are essential for verification. Auxiliary variables

† The verification system was not chosen based on power [LuMc91c], but rather out of convenience, since many people prefer shared memory programming, the global auxiliary system is used as a starting point.
are used to record part of the history of the communication sequence. In order to relate the different interleavings of processes via interleaved communication histories on the global auxiliary variables requires a proof of "non-interference", or a proof that executions in one process do not affect assertions in another and a "satisfaction" proof to show soundness.

In the asynchronous message-passing environment [ScSc84], each pair of processors i and j has 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. Thus, in the actual sending and receiving of a message, $\sigma_{ij}$ and $\rho_{ij}$ are immediately updated and $\rho_{ij} \subseteq \sigma_{ij}$ is invariantly true throughout program execution.

2.2. HAA Proof System and Translation

The logical assertions from the GAA verification environment cannot be directly used as executable assertions, since there is no notion of global variables in a distributed environment. Thus, to evaluate logical assertions containing global auxiliary variables at run time, requires an explicit updating mechanism to be created. The HAA verification proof system [LuSM92a], where HAA is an abbreviation for "History of Auxiliary variables Approach", is a translation from the GAA proof system such that updates of global auxiliary variables are exchanged at communication time. The HAA proof system preserves the properties of the GAA proof system, and it is exactly the property of non-interference from the proof outline in GAA that allows the translation to be performed in a way that is efficient during program execution. Communication and assertion evaluation in the HAA proof outline matches more closely with the operational environment. Hence, the proofs can then be directly transformed into executable assertions to be used in the run-time environment.

Developing the HAA system demands keeping track of which processes communicate with each other. Each process records all updates of their global auxiliary variables with respect to the other processes. When communication occurs between two processes, the processes exchange the updates and also apply the updates locally. This is formalized in the following definitions.

**Definition 2.1:** For a process $\rho_i$, $h_i$ denotes the sequence of all communications that process $\rho_i$ has so far participated in as the receiving process. Thus, $h_i$ is a list consisting of tuples $^\dagger$ representing matching communication pairs of the form

$$[\rho, (\text{Var}, \text{Val}), T, C]$$

where $\rho$ is a process from which $\rho_i$ receives from, $\text{Var}$ is the variable that $\rho$ is transmitting to $\rho_i$ with formal parameter $\text{Val}$. $T$ denotes the time at which the value $\text{Val}$ was assigned to variable $\text{Var}$ and $C$ denotes the communication path.

$^\dagger$ It is important to note that these tuples are different from those defined in the verification system of [Soun84]. All future reference to tuples in this paper will refer to the type defined in Definition
Since there are several processes running in parallel and no concept of a global time, the local time, $T$, is represented by a local instantiation counter that is incremented by one after every execution of a statement. This permits an ordering (time-stamping) for all updates of the GAVs within each process.

To be able to account for the different operations performed on the auxiliary variables, each process has to keep a history of variable updates with respect to the last communication with the other processes. These variable sets are described using the subscript of the corresponding process.

**Definition 2.2:** Let $g_{ij}$ depict the GAV set in process $\rho_i$ with respect to process $\rho_j$, i.e., $g_{ij}$ contains the changes that were made to the GAVs in $\rho_i$ since the last communication with $\rho_j$. $G_i$ is the set of sets $G_{i0}, G_{i1}, \ldots, G_{i(N-1)}$ in process $\rho_i$. Thus, when two processes $\rho_i$ and $\rho_j$ communicate, the values of their respective subsets, $g_{ij} \in G_i$ and $g_{ji} \in G_j$, are exchanged.

When two processes $\rho_i$ and $\rho_j$ communicate, where $\rho_j$ is the sender, $\rho_j$ augments the communication by sending the values of global auxiliary variables that $\rho_j$ updated as well as updates received from other processes between the last and current communications of $\rho_i$ and $\rho_j$. The changes made to the local copies of the global auxiliary variables by $\rho_j$ since the last communication (with any other processor) in $g_{jj}$ are batched together. Each time, before a communication rendezvous, the function $\psi$ applies these changes to all $g_{jk}$’s and after a communication, the function $\varphi$ applies the received copies of the GAVs to the local state of the process.

It can be seen that the so-called “global auxiliary variables” in the HAA system are not really global in the sense that all processes have the same values of the variables at all times. Indeed, it is likely that at the end of the process execution some processes that ran in parallel have different values within their set of GAVs. Non-interference between processes prevents this from being a problem with respect to the proof system.

**2.3. Reliable Communication of State Information**

The HAA proof system provides for direct transformation of assertions from the verification environment into executable assertions for the non-faulty distributed operational environment. Since the HAA system mimics the operational environment, in a system with no faults, assertions are checked by straightforward translation of postconditions, $Q$, into executable assertions. However, in the Byzantine [LaSP82] faulty environment, to operationally test executable assertions, it is necessary to ensure that faulty processors cannot fool executable assertions by incorrect augmented communication of GAV sets through sending inconsistent messages to different processors. The consistency executable assertions strengthen the effectiveness of the

---

2.1. In Section 2.3, the path notion is used.
executable assertions derived from the HAA system. When the value of a variable computed at time \( T \) is communicated to a set of processors on more than one path, there will be two or more tuples in \( h_i \) that satisfy the precondition. Under a bounded number of faults, consistency ensures a non-faulty processor receives a consistent set of input values for its executable assertions.

The notion of consistency does not necessarily have to be explicit; an error-detecting program does not have to explicitly add code to implement consistency. There are classes of problems which are naturally redundant in the problem variables and can be (re)constructed from other variables. For instance, there are types of errors which may occur in some state \( i \), which eventually satisfies properties defined by the intermediate assertions from a verification proof at some later state \( j \), despite the error that occurred at state \( i \). This property returns the system back to a consistent state. Natural redundancy is further discussed in relationship with the class of Branch and Bound problems in Section VI.

2.4. Run-Time Efficiency Considerations

The transformation from the HAA verification environment to the operational environment described above is optimal in the sense that all violations of the program’s specification (in terms of the postconditions on each statement and within the limits of consistency) are caught under a bounded number of faults. However, when considering run-time efficiency, not all of these assertions, nor all of the communicated GAVs are necessary. These two aspects of reducing complexity are treated as follows:

- Assertions involving local variables to a particular process which are necessary in the verification environment are useless in the distributed operational environment. Since the unit of failure and reconfiguration is at the processor level, a processor cannot be trusted to diagnose itself as faulty or fault-free. Thus, assertions using only local variables incur a run-time overhead that is not necessary, and all such assertions can be deleted.

- The fault coverage of certain assertions using the GAVs may be subsumed. Thus, many of the remaining assertions may be removed as well. Likewise, removing some of the assertions may result in certain GAVs no longer being required. Furthermore, certain assertions may be too expensive to evaluate in the operational environment and may be deleted or weakened for that reason.

- If the problem is naturally redundant, it is capable of eventually self-correcting itself. The self-correcting nature may correct errors without using consistency executable assertions. Hence, these assertions are useless and may also be taken out.

III. BRANCH & BOUND ALGORITHMS

The model used to demonstrate the fault-tolerant application-oriented approach of applying executable assertions is the class of Branch and 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 Branch and Bound algorithms use a best-first search strategy. A function used for estimating the cost of each node allows the most promising node to be examined first. This implies that in a sequential environment, the first solution discovered is also the optimal solution. However, in a distributed environment, the first solution found is not necessary the optimal due to the distribution of the tree nodes. Hence, the cost of a solution acts as a global upper bound in the search. The algorithm described in this paper is the N Puzzle Problem where N+1 is a perfect square. The description of the problem is given below.

Figure 3.1: Objective of the N Puzzle Problem (N = 8)

Most people are familiar with the N Puzzle Problem [Quin88]. 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, with 0 representing a blank space as shown in Figure 3.1. The objective is to rearrange the tiles in the minimal number of moves to get from some initial board configuration to a known final configuration. Figure 3.1 shows an example of an initial and final configuration for N=8.

The possible ways of obtaining the final configuration from a given initial configuration is represented by 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.

**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.

\[
\begin{align*}
m_0: b_{u,v} &\leftrightarrow b_{u+1,v} & \text{if } u \neq \sqrt{N+1} \\
m_1: b_{u,v} &\leftrightarrow b_{u-1,v} & \text{if } u \neq 1 \\
m_2: b_{u,v} &\leftrightarrow b_{u,v+1} & \text{if } v \neq \sqrt{N+1}
\end{align*}
\]
Let $M$ be the set consisting of \{m_0, m_1, m_2, m_3\}.

Each node of the tree is referred to as some state, $s_i$, of the puzzle where $i$ is a unique integer. Let $s_0$, the root of the tree, denote the state corresponding to the initial configuration. A path is then the sequence of moves from $s_0$ to some node $s_i$. All states are created by applying a sequence of legal moves to the initial state. These states are then uniquely defined by the path starting from the root and leading to each state. This is formally defined below.

**Definition 3.2:** Let each state, $s_i$, 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 node $s_i$, where the number of moves taken is $k$.

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

where $p_{ij}$ is a move of type $m$, $m \in M$ and $0 < j < k - 1$.

The intermediary states along the same path for $s_i$ with the same starting point are referred to as *path prefixes* of $s_i$. Hence, the notion of reachability is realized when some state $s_i$ and $s_j$ reside along the same path of the search tree.

**Figure 3.2:** An abstraction of the N Puzzle Problem (N = 8).
Definition 3.3: A node denoted by \(s_i\) is reachable from a node denoted by \(s_j\) if \(s_j\) is a path prefix of \(s_i\).

In the N Puzzle Problem, the problem is solved when the initial state is transformed into the final configuration through a minimal number of moves. There may be many such paths leading to a solution in the tree search space or none at all. Definition 3.4 and 3.5 define the search space and the set of solutions.

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

\[
S_1 = \{s_i \mid s_i \text{ is a reachable configuration of } s_0\}
\]

Definition 3.5: If \(s_i \in S_1\) 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, \(A_1\), is the set containing all solutions for the N Puzzle Problem.

\[
A_1 = \{s_i \mid s_i \in S_1 \land s_i \text{ is a solution }\}
\]

Since examining the entire tree is exponential in complexity, 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 \(k\).

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

where \(md_i\) 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 level of the tree. The manhattan distance represents the distance each tile is out of place as well as a lower bound for any solution. The level corresponds to the number of legal moves taken thus far.

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. \(s\) is then an optimal node in the search
**Theorem 3.2:** The optimization function, \( f \) is a monotonic nondecreasing function as \( k \) increases.

**Proof:** Proof by induction on the level of the tree, \( k \). Initially, the optimization function is just the manhattan distance for the starting configuration. Let \( s_i^k \) denote that \( s_i \) exists at level \( k \) of the search tree. The optimization function for \( s_i^k \) is defined as follows:

\[
f(s_i^k) = \text{md}_i + k
\]

The optimization function, \( f \), is monotonically nondecreasing if \( f(s_i^k) \leq f(s_i^{k+1}) \). By Definition 3.1, a legal move either causes the manhattan distance to increase by one or decrease by one. Only one legal move is allowed along any particular path of the tree. If the manhattan distance increases by one, then the function increases by one for both the distance and for the number of moves taken. If the manhattan distance decreases by one then:

\[
\text{md}_{i'} = \text{md}_i - 1
\]

\[
f(s_i^{k+1}) = (\text{md}_i - 1) + (k + 1) = \text{md}_i + k = f(s_i^k)
\]

and the function remains at the same value as before. The decrease in the manhattan distance is compensated by the increase in the number of moves taken, thus providing a nondecreasing monotonic behavior. \( \square \)

By Definition 3.7 and Theorem 3.2, a solution is found when the manhattan distance decreases to zero. The function results in \( k \) for some \( f(s_i) \). This value of \( k \) 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 nodes which possess a bound higher than \( k \) need not be considered. All nodes with cost greater than \( k \) can not possibly lead to a better solution, however, the remaining paths with lower bounds must continue expanding, generating new states until a solution is reached or until the cost exceeds the current upper bound. When all the nodes of the tree have been explored, the solution with the lowest bound is the one which took the minimum number of moves to solve the puzzle. The parallel algorithm described in this paper is based on this concept.

The parallel algorithm involves dividing the work in terms of subtrees and allows many processors to work on the problem simultaneously. A task, executed by a processor, transforms some \( s_i \) to \( s_{i'} \) by all legal moves of type, \( m_k \), where \( m_k \in M \). The algorithm is distributive and autonomous thus requiring only workers to search the solution space. The initial task, \( s_0 \), is assigned to a designated worker to work on. Each task produces up to 4 new tasks according to Definition 3.1. Each worker then retains one task and redistributes the rest to idle workers. When a worker completes the assigned task, other workers are notified so that tasks can be redistributed. The only other communication that occurs among the workers is when a solution, \( s_i \), has been found. This solution is a member of the solution space, \( A_I \), and may not be the optimal
solution, but the cost of the solution can act as an upper bound allowing some pruning to be done. The solution is broadcast to the other workers for updates of each worker’s 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 current solution, $s_{current}$ which has the lowest bound then contains the optimal moves for the puzzle.

The dynamic allocation of tasks avoids the use of a centralized processor. Hence, if a processor should fail, the task that was being worked on is easily redistributed to another processor. The asynchronous nature of the message passing allows workers to work independently without interruptions until an idle worker is acknowledged. The advantage is the decrease in message passing required. The tradeoff is the added difficulty of detecting faults where timing is concerned. This is further discussed in Section VII.

IV. VERIFICATION OF THE N PUZZLE PROBLEM

This section discusses in detail the verification process of the N Puzzle Problem. The annotated algorithm with assertions and assignments to auxiliary variables is presented in Figure 4.3. Not all assertions are listed. There is one assertion, labeled I, (defined in Definition 4.1) that is invariant throughout execution, except during communication. The reason why this assertion is not invariant throughout execution is described later in this section.

The following auxiliary variables are used in the verification proof:

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

From Figure 4.1, there exist two distinct types of processes: a host process and a number of worker processes. The precondition of the host process, denoted by $Pre_H$, assumes that a solution from the initial configuration exists. Therefore, an immediate assertion can be made on $A_i$.

$$Pre_H: \ \ < A_i \neq \emptyset >$$

The postcondition of the host process depends on the value of each worker’s $s_{current}$. At the
{PreH}
Host:
Begin
  S,A = S_I, A_I;
  S,A = ∅, ∅;
  Distribute initial board configuration;
  <S_0 ∪ S_1 ∪ ... ∪ S_{N-1} = S_0>
  Terminate workers;
End;
<PostH>

<PreW>
Worker:
Begin
  Wait for a task to work on;
  s_current = ∞
  While (task > 0)
    Loop
      Work on expanding lowest costing paths;
      If s_current ≠ ∞
        If f(s_task) > f(s_current)
          Discard task.
          T_0 = {s_i | s_i is a reachable configuration of s_task}
          T_1 = {s_i | s_i ∈ T_0 ∧ s_i is a solution state}
          S_i, S, S = S_i − T_0, S − T_0, S ∪ T_1
          If a solution is found
            Notify other workers;
            Update SB_ij for each j;
          If (task > 1) and (worker_j = IDLE)
            T_0 = {s_i | s_i is a reachable configuration of s_task}
            S_i = S_i − T_0
            Distribute task to idle worker_j;
          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 If
      End If
    End While;
    Wait for a task to work on;
    T_0 = {s_i | s_i is a reachable configuration of s_task}
    S_i = S_i ∪ 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 If
  End Loops;
  solution_i = s_current
End.
<PostW>

Figure 4.1: Verification proof outline of the N puzzle problem
termination of the worker processes, each worker’s $s_{current}$ should reflect the same lowest bound. Therefore, the postcondition, Post$_H$, for the host process is:

$$\text{Post}_H: \langle \text{solution}_0 = \text{solution}_1 = \cdots = \text{solution}_{N-1} \land \text{solution}_i \text{ is the optimal cost solution} \rangle$$

The precondition, Pre$_i$, for each worker process $i$ is the assumption that the worker processes initially do not have any tasks to examine and no communication between the workers has occurred. This is represented by

$$\text{Pre}_i: \langle S_i = \emptyset \land SB_{ij} = \emptyset \land RB_{ij} = \emptyset \text{ for all } j, 0 \leq j \leq N - 1, j \neq i \rangle$$

The postcondition, Post$_i$, for each worker, $i$, is the assertion that the current local solution, $s_{current}$ is optimal.

$$\text{Post}_i: \langle s_{current} \text{ is the optimal cost solution} \rangle$$

Since $S$ is the set of nodes and $A$ is the set of solutions that have yet to be examined, $S$ and $A$ are initialized to $S_I$ and $A_I$ respectively in the Host process. Similarly, $S'$ and $A'$ are the set of examined nodes and solutions, respectively, and are initially set to $\emptyset$, since none of the nodes or solutions have been examined.

The distribution of the initial board configuration corresponds to assigning each process, a subset of the nodes in the tree that represents the state space, such that no two processes examine the same set of nodes. This corresponds to partitioning the auxiliary variable $S_I$ into the disjoint sets $S_0, S_1, \ldots, S_{N-1}$. The following assertion is satisfied immediately after the initial board is distributed:

$$\langle S_0 \cup S_1 \cup \cdots \cup S_{N-1} = S_I \rangle$$

Each worker process examines only those nodes that are part of the state space. These nodes are (1) Generated through expansion of other nodes or (2) Received from other processes. The first case implies that each newly generated task must be part of the search space. For the sake of brevity, the details are not shown here, but involve 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 $S_I$. Thus, it can be concluded that the following is always true:

$$\langle s_{\text{task}} \in S_I \rangle$$

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 noting that process $i$ only receives nodes to examine from other processes. As the node is migrated from some process $j$ to process $i$, $\sigma_{ji}$ is immediately updated and $\sigma_{ji} \subseteq S_I$ is true, since all nodes migrated are members of $S_I$. Note that the definition for solutions implies that all solutions are members of $S_I$. In Section II, it was given that $\rho_{ji} \subseteq \sigma_{ji}$ is invariantly true. Hence, the following logically follows:
Since process $i$ examines only those nodes which are part of the search space, the following is also invariantly true:

$$< S_i \subseteq S_t >$$

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 $j$.

In case (1), 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 is $T_0 = \{ s_j \mid s_j \text{ is a reachable configuration of } s_i \}$. This set of elements is deleted from $S_i$ by the following operation: $S_i = S_i - T_0$. This does not violate assertion 4.1 since this new set of nodes is a subset of the original set.

Showing that cases (2) and (3) maintain the truth of $S_i \subseteq S_t$ is part of the satisfaction proof. This requires showing that the communication maintains the truth of the invariant, $S_i \subseteq S_t$. For case (2), process $i$ receives a new task from process $j$. Earlier, it was shown that $\rho_{ij} \subseteq S_t$, hence, $S_i$ is updated by first determining the set of nodes associated with the subtree for which the received node is the root. This is done by determining all the reachable configurations from the received nodes as defined in Definition 3.3. The set of nodes is then added to $S_i$. Since the received node is a member of $S_t$, all the reachable nodes from the received node are also members of $S_t$. Hence, the truth of $S_i \subseteq S_t$ is maintained. A similar argument can be made for case (3).

The task of examining each node in the state space is assigned to a process. This is represented by the following:

$$< S_0 \cup \cdots \cup S_{N-1} \cup \{ s \mid s \in \sigma_{ij} - \rho_{ij} \wedge \text{s is a task, where } 0 \leq i, j \leq N - 1, i \neq j \} = S >$$  \hspace{1cm} (4.2)

Task migration does not change the set of nodes to be examined as a whole. Instead, task migration only changes the set of nodes to be examined by the migrating and receiving worker processes. However, because of the asynchronous nature of the algorithm, it is possible that process $i$ sends a task to a process $j$, which is not ready to receive the task. Therefore, it is necessary to include the following set in the above assertion.

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

It is also necessary to ensure that the algorithm examines only those nodes which are in the state space and does not discard any tasks or solutions before they are examined. This can be ensured by making the following invariantly true:

$$< (S' \cup S = S_t) \wedge (A' \cup A = A_t) >$$  \hspace{1cm} (4.3)
The truth of this can be seen by observing that (1) The assertion stated in (4.3) is true at the beginning of the worker process’ program execution and (2) \( S, S', A \) and \( A' \) are updated when a subtree is pruned from the search space. The updating is done by finding the set of nodes associated with the subtree and determining which nodes can be pruned and which are possible solution nodes. This set of nodes can be found by finding all reachable nodes from the root of the subtree. The set of all nodes associated with the pruned subtree is then deleted from \( S \) and added to \( S' \). Simultaneously, the solution nodes are deleted from \( A \) and added to \( A' \). It is obvious, now, that the assertion stated in (4.3) is always true, since, the assertion was initially true, and remains true even after the updates occur.

In each process \( i \), \( s_{current} \) is the solution that is known by process \( i \) to have the lowest bound. Changes to \( s_{current} \) are made as more information about other solutions become known by other processes. Therefore, the following assertion is another invariant:

\[
<s_{current} = \min(a \mid a \in R_i, \text{ where } R_i = \bigcup RB_{ij})>
\] (4.4)

Since this is a distributed environment, the set of solutions received by each process upon termination of the program should be equivalent. The following assertions ensure this. The assertions must be invariantly true with the exception of when a solution is currently being broadcast. This is because the auxiliary variables are updated only after the broadcasted solution is received.

\[
<SB_{ij} = R_j \bigcup \{x \mid x \in \sigma_{ij} - \rho_{ij} \land x \text{ is a solution}\}>
\] (4.5)

\[
<SB_{i0} = \cdots = SB_{iN-1}>
\] (4.6)

The assertion stated in (4.5) states that the solutions sent by process \( i \) to process \( j \) are the same as those received by process \( j \) from process \( i \), except for the ones still in transit. This implies that the assertion stated in (4.6) is also 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 a lesser cost. Mathematically, this can be described as follows:

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

(4.7)

**Definition 4.1:** Let the assertion \( I \) denote the conjunction of the logical expressions expressed in (4.1)-(4.7).

This assertion, \( I \), remains true throughout the program execution with the exceptions stated for (4.5) and (4.6). These exceptions are immediately rectified after communication through the assignments to \( S_{ij} \).

The algorithm consists of two additional loop invariants, an inner loop and an outer loop, as depicted in Figure 4.1. The inner loop invariant for some process \( i \) is defined as follows:
< (task > 0 ∧ S_i ≠ ∅) ∨ (task = 0 ∧ S_i = ∅) > \tag{4.8}

The outer loop invariant for worker process i is then:

< ((worker_i = busy ∧ S_i ≠ ∅) ∨ (worker_i = idle ∧ S_i = ∅ ∧ σ_ij − ρ_ij = ∅)) > \tag{4.9}

Upon termination of the Host process, each worker process should have knowledge of the optimal solution. This requires the termination of the worker process in Figure 3.3 to imply that each process received the same set of broadcast solutions.

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

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

**Proof:** The outer loop invariant and termination of all worker processes implies that σ_ij − ρ_ij = ∅. Hence, for a process i and any worker process j, SB_{ij} = RB_{ij}. From (4.7), it can be concluded that at program termination, RB_{0i} = R_{1i} = \ldots = R_{N-1i} for any worker process i. Hence, for any worker process j, R_j = \bigcup RB_{ji} which implies:

\[ R_0 = \ldots = R_{N-1} \]

Lemma 4.1 and Theorem 3.2 can now be used to show that the termination of the program implies that each worker processor has found the optimal solution.

**Theorem 4.1:** *The termination of the program implies the postcondition, Post_{II}*

**Proof:** At the termination of the outer loop, all processes are idle and hence, for all i, where 0 ≤ i ≤ N − 1, the set of nodes to be examined by worker i is empty: S_i = ∅. Since, the verification proof shows that S_0 ⊆ S_1 ⊆ \ldots ⊆ S_{N-1} = S, it can be concluded that S_i is also empty: S = ∅. Furthermore, A ⊆ S and A ⊆ A', which implies A' = A'. Therefore, from (4.4), (4.7) and Lemma 4.1, it can be easily concluded that s_{current} is the same for each process and is the minimal cost as defined by the optimization function of Definition 3.7. From Lemma 4.1 and Theorem 3.2, this minimal cost is thus the optimal cost. □

For the translation to HAA, the augmented communication primitives φ and ψ are simply added to each communication followed by consistency on the received data tuples. However, by relaxing strict consistency, a great deal of efficiency improvements can be made by weakening some of the assertions. Efficiency issues are discussed in the next section.
V. EFFICIENCY

The executable assertions are derived directly from the verification proof and are executed after the receipt of a message. A summary is presented in Table 5.1.

<table>
<thead>
<tr>
<th>Transformation of Assertions</th>
<th>Verification Assertion</th>
<th>Executable Assertion</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ọọ_Ọọ : s_i ∈ S_i</td>
<td>Directly from S_i ⊆ S_i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ọọ_Ọọ : s_i ∈ A_i (assuming that s_i is a proposed solution)</td>
<td>The availability of s_i induces an element by element set membership test from</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ọọ_Ọọ : s_current = min (a</td>
<td>a ∈ R_i)</td>
<td>Direct result from s_current = min (a</td>
<td>a ∈ R_i)</td>
</tr>
<tr>
<td>Ọọ_Ọọ : (R_ji ⊆ S_j)</td>
<td>From SB_ij = R_ji ∪ {x</td>
<td>x ∈ σ_ji − ρ_ij ∧ x is a solution} ∧ SB_i0 = ⋯ = SB_iN-1 ∧</td>
<td></td>
</tr>
<tr>
<td>Ọọ_Ọọ : cost_0 = f(s_i), where cost_0 is the received cost of s_i</td>
<td>Result of weakening the condition: For all x ∈ A, there is an i and y such that y ∈ R_i and f(y) ≤ f(x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ọọ_Ọọ : f(s_i) &gt; f(s_prev), where s_prev is the previously proposed solution</td>
<td>Result of weakening the condition: For all x ∈ A, there is an i and y such that y ∈ R_i and f(y) ≤ f(x) to consider only proposed (available) moves</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program Postcondition, PostH:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;solution_0 = ⋯ solution_{N-1} ∧ solution_i is optimal solution&gt;</td>
<td>( \Omega_{P_H} : \text{solution}<em>0 = ⋯ \text{solution}</em>{N-1} )</td>
<td>Directly from the postcondition.</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Summary of Executable Assertions

Let s_i denote a received task/solution. A subset of these executable assertions are those directly derived from the invariant. The first part of the invariant is the following:
As mentioned in Section II, the transformation from an HAA proof to a set of run-time assertions can be done algorithmically, however, this results in fault-tolerant programs in which the run-times are significantly increased compared to the non fault-tolerant version. This trade-off between speed and reliability is not justifiable for many applications. Therefore, it becomes necessary to examine heuristic means for reducing the execution run-time. The heuristics are based on choosing a subset of the assertions using global auxiliary variables.

In the previous section, it was observed that the nodes to be examined, those in $S_i$, changed as a result of either pruning, receiving a new task or the migration of a task. As a direct consequence, there are two types of fault occurrences. If process $i$ is faulty then the wrong subtree may be pruned. The objective of finding the optimal solution is not affected if the pruned subtree does not contain the path leading to an optimal solution. The executable assertion that can catch the fault causing the optimal solution to be pruned off is discussed later. The other type of fault occurs when a task is communicated erroneously. As a result, process $i$ either receives an incorrectly communicated task, or incorrectly migrates a task. Therefore, it becomes feasible for every receiving process to check whether the task received represents a node in the search space. From Definition 3.2, a migrated task, $s_i$, 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 by first applying the move $p_{i0}$ to the initial configuration, as defined in Definition 3.1. Assuming that head($s_i$) extracts the first element in the list that $s_i$ represents and that tail($s_i$) extracts everything but the first element from $s_i$, the pseudocode for carrying out this check is as follows:

It is possible to have a task that was incorrectly communicated represent a node in the search space. As long as the node is the root of a subtree which does not contain the optimal solution, the search is not affected. The executable assertion which handles the case when the node is the root of a subtree that does contain the optimal solution is discussed later.

For the second part of the invariant assertion, a weaker condition is used instead, because use of auxiliary variables in which no one process has a complete knowledge poses implementation problems.

$$< S_1 \cup S = S_f \land A \cup A = A_f >$$

The pseudocode of Figure 5.1 can be used to test each migrated node or solution to see whether that node belongs to the search space.
If \( p_{i0} \) is applicable to \( s_0 \) then

apply \( p_{i0} \) to \( s_0 \) and assign to \( n \)

else

ERROR;

\( s_i = \text{tail}(s_i) \);

while \( s_i \) is not empty

if head\((s_i)\) is applicable to \( n \) then

apply head\((s_i)\) to \( n \) and assign to \( n \)

\( s_i = \text{tail}(s_i) \)

else

ERROR

end while;

**Figure 5.1:** \( \Omega_{E_1} \): Pseudocode to check whether a sequence of moves is legal

In addition, it is desirable to check whether a solution node is really a solution. Assume that \( \text{goal}_{u,v} \) 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 is made into an executable assertion which checks whether the received solution node is really a member of \( A_t \). The following pseudocode can be used to implement this executable assertion:

```plaintext
For (u=1; u<=\sqrt{N+1}; u++)
  For (v=1; v<=\sqrt{N+1}; v++)
    If (t_{u,v} == \text{goal}_{u,v}) then ERROR;
```

**Figure 5.2:** \( \Omega_{E_2} \): Check to see whether a proposed solution is really a solution.

The third part of the invariant is stated as follows:

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

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

\[ \Omega_{E_3}: \text{If } s_{\text{current}} != \min (a | a \in R_i) \text{ then ERROR;} \]

**Figure 5.3:** \( \Omega_{E_3} \)

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 into an executable assertion for the run-time environment because of the difficulty in 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 piggy-backed with solutions as $[k,R_{ji},T,P]$ and $[k,SB_{ji},T,P]$, respectively, then on receipt of the solutions, the following pseudocode is executed:

| $\Omega_{Fi}$: | If $\neg(R_{ji} \subseteq SB_{ij})$ then ERROR; |

**Figure 5.4: $\Omega_{Fi}$**

The last assertion

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

is also carried out using a weaker condition. The above implies that for a process $i$, the successive values of $s_{current}$ 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 $s_{prev}$ in the pseudocode) are 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:

| $\Omega_{Fp}$: | IF $s_i$ is a solution then |
| | IF $s_i > s_{prev}$ then ERROR; |

**Figure 5.5: $\Omega_{Fp}$**

It is not enough to obtain a solution with a lower cost, but also desirable to ensure that the 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.

| $\Omega_{F5}$: | Let $\text{cost}_1 = f(s_i)$; |
| | If $\text{cost}_0 \neq \text{cost}_1$ then ERROR; |

**Figure 5.6: $\Omega_{F5}$**

The executable assertions denoted by $\Omega_{F1}$, $\Omega_{F2}$, $\Omega_{F3}$, $\Omega_{F4}$, and $\Omega_{F5}$ measure the validity of the information. Hence, these executable assertions fall into the feasibility class of executable
assertions. $\Omega_p$ is based on the goal of finding the solution with the least moves. Each solution received should exhibit a better way of solving the puzzle than the previous. Therefore, $\Omega_p$ is an example of a progress constraint. Note that $\Omega_{F_1}$ and $\Omega_{F_3}$ are applied to migrated tasks and solutions, while $\Omega_{F_2}$, $\Omega_{F_3}$, and $\Omega_{F_4}$ and $\Omega_p$ are applied to broadcast solutions.

Earlier, cases were discussed where the optimal solution was pruned off incorrectly. This would imply that the postcondition of the program, Post_H, is false. The following assertion tests the postcondition.

\[
\text{if } \forall [i, s_{\text{current}}, T, P] \neq s_{\text{current}} \text{ then } \text{ERROR!}
\]

**Figure 5.7**: $\Omega_{F_1}$: Testing the postcondition

This section has described the weakening of the verification proof-based assertions to create an efficient fault-tolerant algorithm. The algorithm annotated with the executable assertions derived in this section appears in Figure 5.8.

**VI. CONSISTENCY FROM NATURAL REDUNDANCY**

The importance of consistency, or ensuring that two non-faulty processors executing the same executable assertion have the same result, was mentioned in Section II. This section applies the concept of consistency to the executable assertions of the model problem.

In the algorithm, it is possible for a subtree with the optimal solution to be improperly communicated or never communicated at all. If a solution is improperly communicated, the executable assertions of the receiving processes will catch the fault. However, if the solution is never broadcast, then the other processes will not be able to detect this. This case is coined the “silent worker” scenario. Generally, a way of testing the postcondition would be for each process to broadcast its perception of the best solution to all other processes. The consistency condition can then be ensured by using the Byzantine General’s algorithm [LaSP82]. This is clearly infeasible due to the time overhead and connectivity requirements. This section introduces the notion of natural redundancy and demonstrates how this concept implies consistency.

A naturally redundant algorithm [LaMG91] running on a processor architecture P has at least the potential to restore the correct value of any single erroneous component in its output. 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 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
**Worker:**

Begin

\[
\forall \text{ Wait for a task to work on } \phi;
\]

\[s_{\text{current}} = \infty\]

Call \( \Omega_{\phi_{1}}(\text{task}); \)

Call \( \Omega_{\phi_{5}}(\text{task}); \)

Loop

While ( task > 0)

Work on expanding lowest costing paths;

If \( s_{\text{current}} \neq \infty \)

If \( f(s_{\text{task}}) > f(s_{\text{current}}) \)

Discard task.

\[ T_{0} = \{ s_{i} \mid s_{i} \text{ is a reachable configuration of } s_{\text{task}} \} \]

\[ T_{1} = \{ s_{i} \mid s_{i} \in T_{0} \land s_{i} \text{ is a solution state} \} \]

\[ S_{i}, S, S' = S_{i} - T_{0}, S - T_{0}, S' \cap T_{1} \]

If a solution is found

\[ \forall \text{ Notify other workers } \phi; \]

Update \( SB_{ij} \) for each \( j \);

If (task > 1) and (worker\(_{i} = \text{IDLE})

\[ T_{0} = \{ s_{i} \mid s_{i} \text{ is a reachable configuration of } s_{\text{task}} \} \]

\[ S_{i} = S_{i} - T_{0} \]

\[ \forall \text{ Distribute task to idle worker } \phi; \]

\[ \forall \text{ If a solution is reported } \phi \]

If \( f(s_{\text{recv}}) < f(s_{\text{current}}) \)

Call \( \Omega_{\phi_{2}}(\text{solution}); \)

Call \( \Omega_{\phi_{3}}(\text{solution}); \)

Call \( \Omega_{\phi_{4}}(\text{solution}); \)

Call \( \Omega_{\phi}(\text{solution}); \)

\[ s_{\text{current}} = s_{\text{recv}}; \]

Update \( R_{ij} \), where \( j \) is the sending process;

End While;

\[ \forall \text{ Wait for a task to work on } \phi; \]

\[ T_{0} = \{ s_{i} \mid s_{i} \text{ is a reachable configuration of } s_{\text{task}} \} \]

\[ S_{i} = S_{i} \cup T_{0} \]

Call \( \Omega_{\phi_{1}}(\text{task}); \)

Call \( \Omega_{\phi_{5}}(\text{task}); \)

\[ \forall \text{ If a solution is reported } \phi \]

If \( f(s_{\text{recv}}) < f(s_{\text{current}}) \)

\[ s_{\text{current}} = s_{\text{recv}}; \]

Call \( \Omega_{\phi_{2}}(\text{solution}); \)

Call \( \Omega_{\phi_{3}}(\text{solution}); \)

Call \( \Omega_{\phi_{4}}(\text{solution}); \)

Call \( \Omega_{\phi}(\text{solution}); \)

Update \( R_{ij} \), where \( j \) is the sending process;

End Loop;

\[ \text{solution}_{i} = s_{\text{current}} \]

End.

Call \( \Omega_{\phi_{3}} \)

**Figure 5.8:** Fault-Tolerant Algorithm In HAA.
divided into phases that are themselves naturally redundant

An algorithm may be be loosely correct [LaMG91] if the value of a component of the output of a phase 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).

It is now shown that the N Puzzle algorithm is phase-wise naturally redundant in the loose sense with respect to the broadcast of the solutions, i.e., the broadcast solutions are the output of each phase, where each phase is defined as being between broadcast 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.

**Theorem 6.1:** The N Puzzle algorithm of Section III annotated with the executable assertions developed in the Section V and displayed in Figure 5.8 is a phase-wise naturally redundant algorithm in a loose sense.

To show that the algorithm is a phase-wise naturally redundant algorithm in a loose sense requires showing that 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 which 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 its 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, worker A’s solution will never be considered.

**Figure 6.1** 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 it. 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 worker A’s current bound. Other workers then update their knowledge of the current best bound as well.

![Diagram of工人A的Bound](image)

**Figure 6.2** (a.) The bound received by worker A is lower than that of all other workers. (b.) All workers self-correct themselves when a lower bound is discovered.

2: Worker A completes all tasks assigned before a better solution is found, in which case, worker A’s result will be the optimal solution to the problem.

![Diagram ofWorkerA完成任务](image)

**Figure 6.3** Bound received by worker A is lower than that of all other workers at the completion of the initial stage.

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 initial stage to check with the others. Hence, consistency is inherent in the algorithm and no additional constraints are necessary at this point.

**Case 3:** Suppose that a worker continues to work while all other workers are now idle. All other workers will request that the busy worker distribute some work to the idle workers. If there is no response, then that worker has only one job. A timeout value \( t \) can then be determined
dynamically by deducing that if the busy worker has only one job, $t$ can be no longer than the

time it takes to expand the current best solution plus some delay in message passing. If it is, then
the current best is already found and it is no longer necessary to wait on that worker. The time to
expand the current best solution, $s_{\text{current}}$ at level $n$, is just $n$ times the time to expand a single state.

The only other consideration needed is to ensure that all likely solutions have been
searched. The assertions developed so far only apply to information passed at the conversation
boundaries of the process. Therefore, information which is not passed cannot be detected. In a
Byzantine environment, a processor or worker could in effect find the optimal solution and never
broadcast it to the other workers. This would make the other processors’ global view of the sys-
tem inaccurate. This silent worker scenario is handled by means of an additional final stage.

**Case 4:** The silent worker scenario occurs when the optimal solution has been migrated to a
worker who remains silent and doesn’t report it to the other workers. Thus, at the end of the ini-
tial stage, a solution is chosen, $s_{\text{current}}$, which may not be the best solution in $A$. The final stage
allows the system to re-examine the search space and confirm that the result received in the ini-
tial stage is the optimal solution.

---

$\Omega_c$: Redistribute the puzzle among the workers;

$S_{\text{best}} = S_{\text{current}}$;

Call Worker with $S_{\text{best}}$ as the initial best bound:

If $S_{\text{best}} \neq S_{\text{current}}$ then

ERROR!

Figure 6.5 $\Omega_c$

---

The final 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 nec-
essary in order to keep a silent worker from retaining the optimal solution again. However, after
the initial stage, a solution has already been found which has a bound lower than most paths in
the tree. Therefore, many paths need not be considered in the final stage if the bound exceeds the
previous optimal bound. There may be more than one silent worker, thus the number of final
rounds necessary depends on the upper bound on the number of silent workers allowed.

**Theorem 6.2:** In the presence of $k-1$ faults, where $k$ is the number of disjoint subsets, the optimal
solution is found after $k-1$ final rounds.

**Proof:** Theorem 6.2 can be proven by induction on the number of disjoint sets. Let the base case
be $k = 2$ sets of workers, and let these sets be denoted as set A and B. Show that the maximum
number of faulty workers allowed is $k - 1$. Initially, the root of the tree to be searched will be
divided into two subtrees. Set A will search one subtree while set B searches the other. Assuming the worst possible case, let a worker in set A hold the optimal solution and remain silent about it. At the end of the initial stage, a solution \( S_{\text{best}} \) is chosen as the optimal solution not knowing that a better solution exists within the silent worker. At the final stage, the two initial subtrees are swapped such that set A now works on the subtree that was solved by set B. However, both workers need not search the entire tree because an upper bound was already found from the initial stage. Since the sets are disjoint, there is no chance for the silent worker to access the path containing the optimal solution. By the end of the final stage, a new \( S_{\text{best}} \) will be discovered which is less than the previous solution. In this case only one faulty worker is allowed with one final round and two subsets.

If two faulty workers were allowed, it is possible that the optimal solution falls in the hands of both faulty workers each time the problem is redistributed. Hence more than \( k-1 \) faulty workers does not guarantee that the optimal solution can be revealed.

Now, assume that for \( k = n \) sets of disjoint workers, up to \( n - 1 \) faulty workers are allowed. Show that for \( n + 1 \) sets of disjoint workers, \( n \) faulty workers can still guarantee that the optimal solution will be found. Let the first \( n \) subtrees be distributed to the the first worker in each set. If the optimal solution is located in the \( 1^{\text{st}} \) subtree, then after \( n \) rounds, each worker remains silent about it. However, at the \( (n + 1)^{\text{st}} \) round, the first subtree will be distributed to the only non-faulty worker who will reveal a solution whose cost is lower than all previously reported solutions. Hence, for \( n+1 \) disjoint sets of workers, at least one set must be totally non-faulty.

It would seem that more faults can be tolerated by maximizing the number of disjoint sets, but this is not the most efficient way to implement. The more disjoint sets there are, the more degraded the system becomes. The reason is due to the limitations in job migrations. If a worker within a disjoint set becomes idle, it can request another worker within its set to distribute a task. However, if all workers within a set become idle, they will all remain idle until the next search round. If one subtree is fairly small compared to the rest of the tree, it is likely for one set of workers to remain idle for quite some time. This is not an efficient utilization of the processors. Nonetheless, without the sets being disjoint, it is not guaranteed that the optimal path does not migrate back to the silent worker again during a final stage.

This section has shown that the the fault-tolerant program of Figure 5.8 is naturally redundant. Hence, there is no need for an explicit consistency constraint predicate. Instead, a final stage is included to mask out the effects of a silent worker.

**VII. TRAVELING SALESMAN PROBLEM**

The Traveling Salesman problem is a more realistic Branch and Bound algorithm than the N Puzzle problem, and involves finding a minimal way of visiting each city once. Each route between cities corresponds with a cost. The process of optimizing the path to every city is very similar to the N-puzzle problem except a specific path from one city to another is considered
each time, rather than a set of moves. The path information is stored in a cost adjacency matrix and an optimization function determines whether to include or not include a particular path. Like all Branch and Bound problems, this optimization function increases monotonically each time a path is considered. Once a complete path of all cities is found, the cost associated with the search becomes the proposed solution.

Both the N Puzzle and the Traveling Salesman require the use of an optimization function to choose the next state to expand. The optimization function used in the Traveling Salesman algorithm is the one given by [LMSK63]. Furthermore, both optimization functions exhibit similar monotonic behavior. The only difference is in the definition of a state. In the N-Puzzle problem, a state was a particular configuration of a board. For the Traveling Salesman problem, a state consists of an adjacency matrix describing the paths which are or are not included in the tour.

In developing the fault-tolerant version of the Traveling Salesman, the Changeling methodology is repeated. Since consistency was defined over the class of Branch and Bound algorithms it remains the same for the Traveling Salesman problem.

In the verification, the proposed solutions are examined. Each solution has bounds on the variables associated with the cost of the path. Assertions relate that the number of paths included in the tour must equal the number of cities; otherwise, a city would be visited more than once. Furthermore, the cost of the proposed solution is exactly the sum of the initial paths of the cities included in the tour.

The next section discusses some performance issues regarding the Traveling Salesman problem annotated with executable assertions and the consistency condition.

**VIII. A MODEL FOR PERFORMANCE**

Performance is one of the key issues when evaluating a fault-tolerant algorithm. It is obvious that to make a system fault-tolerant, some cost of overhead time cannot be avoided, but a good fault-tolerant system will try to minimize the overhead and offer maximal fault coverage. For most fault-tolerant systems, there is a tradeoff between the cost of providing fault tolerance and the number of faults covered. Application-Oriented Fault Tolerance attempts to eliminate the cost of overhead by using executable assertions rather than explicit redundancy yet still provide maximal fault coverage. In this section, a theoretical performance model for Branch and Bound problems is proposed and validated. To evaluate the efficiency of using this approach, a performance comparison is made to the alternative approach of explicit replication.

**Lemma 8.1:** The time for the fault-tolerant algorithm with no faults occurring is:

\[ t_{nf} = t_{init} + t_{final} \]  \hspace{1cm} (8.1)
Proof: According to the algorithm listed in Figure 5.8, the fault-tolerant algorithm consists of two stages which execute consecutively. The initial stage, $t_{init}$, measures the time to examine the search tree and propose a solution for the problem. The time for the final stage, $t_{final}$, is the time to verify the postcondition of all the worker processes by re-examining the search space confirming that the proposed solution is the optimal solution using the upper bound proposed in the initial stage. The run-time of the fault-tolerant algorithm can then be represented as the summation of the two stages.

For comparison purposes, the fault-tolerant algorithm’s performance can be expressed in terms of the original Branch and Bound algorithm’s performance. Let the execution of the original algorithm be denoted by $t_{orig}$. In the initial stage, the executable assertions derived from the HAA intermediate assertions impose an additional overhead on the original algorithm. Partitioning of the search space also causes some overhead to be incurred. Let the overhead imposed by the assertions and the partitioning be denoted by $t_{assert}$. The run-time of the initial stage of the fault-tolerant algorithm can then be rewritten as the summation of $t_{orig}$ and $t_{assert}$.

When no faults occur, the solution found at the end of the initial stage is the optimal solution. All states with cost less than the upper bound in the final stage, will then be re-examined to assure that the optimal solution has been revealed and is not hidden by a silent worker. Therefore, the time to complete the final stage is a fraction of the time to complete the initial stage and is dependent on the amount of the search tree pruned in the initial stage.

**Definition 8.1** The pruning factor, $p$, is defined as follows:

$$
p = \begin{cases} 
1 & \text{if } t_{final} = 0 \\
1 - \frac{t_{final}}{t_{init}} & \text{otherwise}
\end{cases}
$$

(8.2)

The pruning factor of a tree reflects the number of nodes that were discarded as a result of finding a solution. It is dependent on the location of the optimal solution and the time when it is discovered relative to other potential solution nodes. A large pruning factor, $p$, suggest that the proposed solution occurred near the top of the tree which allowed the majority of the nodes in the search space to be rejected as possible solutions. As a result, only a few nodes are re-examined in the final stage. On the other hand, if the pruning factor is small, it is possible for the final stage to virtually mirror the initial stage in the number of nodes to examine. The final stage time, is, therefore, a fraction of the initial stage time where the pruning factor defines the percentage of the tree pruned.

The final stage consists of $n$ rounds for $n$ faults, due to the partitioning of the workers as described in Theorem 6.2. The performance of the fault-tolerant algorithm in the presence of no faults is described by Theorem 8.1.
**Theorem 8.1** The fault-tolerant algorithm’s performance in the presence of no faults is defined in terms of the original algorithm’s performance and $p$, where $n$ is the number of faults tolerable.

\[ t_{nf} = (t_{orig} + t_{assert}) + (1 - p)[t_{orig} + t_{assert}]n \]  

**Proof:** By the discussion above and by replacing $t_{init}$ of Lemma 8.1, with $t_{orig}$ and $t_{assert}$, and applying the pruning factor, $p$, given in Definition 8.1, Theorem 8.1 immediately follows. \( \Box \)

The model can now be extended to take into account the additional run-time incurred when faults are present. Faults can cause a delay in finding a solution since a proposed solution may be suppressed resulting in more of the search tree to be explored. However, Branch and Bound is a naturally redundant algorithm and has the potential to recover from a fault. This delay in finding the solution is represented by $t_{delay}$. Faults can also cause the bound used in the final stage to be higher than the optimal solution, resulting in an increase in the amount of time required to re-examine a larger subtree. This additional increase in search space for the final stage is denoted by the ratio $e$. Incorporation of these two variables into the model allows the overhead caused by the fault to be isolated.

**Lemma 8.2:** The performance model for a fault-tolerant algorithm with faults is represented by the following:

\[ t_f = (t_{orig} + t_{assert} + t_{delay}) + (1 - p + e)(t_{orig} + t_{assert})n \]  

where \((p - e) > 0\)

**Proof:** A fault may cause an increase in the time to propose a solution. This, in turn, causes more of the search tree to be examined. If the fault also suppresses the optimal solution, the subtree to re-examine is enlarged and reflected in the variable, $e$, where $e$ ranges from 0 to $p$. If $e$ is 0, then the fault does not influence the number of nodes to be re-examined and $t_{final}$ is the same as the case with no faults. However, if $e$ is equal to $p$, then the search time for the final stage will be the same as the initial stage. An increase in $e$ results in a decrease in performance which confirms what is expected. \( \Box \)

**8.1 Efficiency Comparisons**

To compare two parallel algorithms requires a look at their respective efficiencies, the amount of work each processor contributes to the total speed-up of the algorithm. For assessment purposes, the classic error-detecting method of duplex resource replication is chosen for comparison with the Application-Oriented Fault-Tolerant approach. In a duplex system, each processor’s hardware is replicated and two copies of the algorithm are executed simultaneously on this hardware. A comparator is added between each pair of processors for error detection. There is no overhead in run-time for the duplex system.

The efficiency for a duplex system is defined as the speedup divided by the number of processors replicated $n + 1$ times, where $n$ is the number of faults tolerable in the system.
Definition 8.2 For the proposed model, efficiency is defined as the speed-up over the number of processors:

\[ E_{ao} = \frac{t_{seq}}{t_f} \]  

(8.5)

where speed-up is just the sequential time, \( t_{seq} \), over the parallel time, \( t_f \), taking faults into consideration.

\[ E_{ao} = \frac{t_{seq}}{(t_{orig} + t_{assert} + t_{delay}) + (1 - p + e)(t_{orig} + t_{assert})n} \]  

no. processors

(8.6)

A comparison of the two efficiency expressions shows that there exist a range of values for \( t_f \) such that \( E_{ao} > E_{duplex} \).

Theorem 8.2: Any instance of a problem, where the delay of the initial stage caused by a fault is bounded by the following, gives better efficiency results for the Application-Oriented approach.

\[ t_{delay} < n [(p - e)t_{orig} - (1 - p + e)t_{assert}] - t_{assert} \quad \text{where} \quad (p - e) > 0 \]  

(8.8)

Proof: The proof of Theorem 8.2 follows from comparing \( E_{ao} \) and \( E_{duplex} \) and finding the crossover points where the efficiencies of the two approaches intersect. □

This range of values for \( t_{delay} \) and \( e \) is dependent on the solution space, the pruning factor and the partitioning effect on the original algorithm. If \( t_{delay} \) is small, the proposed fault-tolerant algorithm performs better than the original algorithm executing in a duplex system. This implies that problem instances with a high pruning factor and with little overhead due to partitioning, are more likely to effectively utilize the processors in a system using the Application-Oriented approach.

8.2 Validation of the Performance Model

To validate the performance model, the original and fault-tolerant algorithms were implemented on an Intel iPSC/2 for a 35 city traveling salesman problem. Run times and pruning factors for this problem are listed in Table 8.1 for the fault-tolerant algorithm when no faults occur. The pruning factor varies with the number of processors since different parts of the search tree are examined due to the dynamic distribution of the tasks.

In a faulty environment, the following relationship is true by Lemma 8.1 and 8.2:

\[ t_{orig} < t_{nf} \leq t_f \]  

(8.9)
<table>
<thead>
<tr>
<th>No. of Proc.</th>
<th>( t_{\text{initial}} ) (sec)</th>
<th>( t_{\text{final}} ) (sec)</th>
<th>( t_{\text{total}} ) (sec)</th>
<th>( t_{\text{delay}} ) (sec)</th>
<th>( p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3480</td>
<td>2386</td>
<td>5866</td>
<td>.31</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3430</td>
<td>2348</td>
<td>5778</td>
<td>.32</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1942</td>
<td>988</td>
<td>2930</td>
<td>.49</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>660</td>
<td>421</td>
<td>1081</td>
<td>.36</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>640</td>
<td>220</td>
<td>860</td>
<td>.66</td>
<td></td>
</tr>
</tbody>
</table>

**Table 8.1:** Representative run-times for the fault-tolerant algorithm in a fault-free environment.

<table>
<thead>
<tr>
<th>Worker</th>
<th>( t_{\text{initial}} ) (sec)</th>
<th>( t_{\text{final}} ) (sec)</th>
<th>( t_{\text{total}} ) (sec)</th>
<th>( t_{\text{delay}} ) (sec)</th>
<th>( e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>560</td>
<td>228</td>
<td>788</td>
<td>-80</td>
<td>1.340x10^{-2}</td>
</tr>
<tr>
<td>1</td>
<td>794</td>
<td>228</td>
<td>1021</td>
<td>154</td>
<td>1.255x10^{-2}</td>
</tr>
<tr>
<td>2</td>
<td>803</td>
<td>226</td>
<td>1028</td>
<td>163</td>
<td>9.013x10^{-3}</td>
</tr>
<tr>
<td>3</td>
<td>503</td>
<td>228</td>
<td>731</td>
<td>-137</td>
<td>1.226x10^{-2}</td>
</tr>
<tr>
<td>4</td>
<td>694</td>
<td>228</td>
<td>922</td>
<td>54</td>
<td>1.340x10^{-2}</td>
</tr>
<tr>
<td>5</td>
<td>798</td>
<td>220</td>
<td>1018</td>
<td>158</td>
<td>-3.988x10^{-4}</td>
</tr>
<tr>
<td>6</td>
<td>708</td>
<td>226</td>
<td>934</td>
<td>66</td>
<td>1.365x10^{-2}</td>
</tr>
<tr>
<td>7</td>
<td>698</td>
<td>228</td>
<td>925</td>
<td>58</td>
<td>1.156x10^{-2}</td>
</tr>
<tr>
<td>8</td>
<td>790</td>
<td>695</td>
<td>1486</td>
<td>151</td>
<td>7.438x10^{-1}</td>
</tr>
<tr>
<td>9</td>
<td>728</td>
<td>259</td>
<td>987</td>
<td>89</td>
<td>6.053x10^{-2}</td>
</tr>
<tr>
<td>10</td>
<td>654</td>
<td>227</td>
<td>880</td>
<td>14</td>
<td>1.073x10^{-2}</td>
</tr>
<tr>
<td>11</td>
<td>715</td>
<td>216</td>
<td>931</td>
<td>75</td>
<td>-5.196x10^{-3}</td>
</tr>
<tr>
<td>12</td>
<td>679</td>
<td>251</td>
<td>930</td>
<td>39</td>
<td>4.899x10^{-2}</td>
</tr>
<tr>
<td>13</td>
<td>436</td>
<td>228</td>
<td>664</td>
<td>-204</td>
<td>1.275x10^{-2}</td>
</tr>
<tr>
<td>14</td>
<td>803</td>
<td>228</td>
<td>1031</td>
<td>163</td>
<td>1.315x10^{-2}</td>
</tr>
<tr>
<td>15</td>
<td>1241</td>
<td>290</td>
<td>1531</td>
<td>601</td>
<td>1.092x10^{-1}</td>
</tr>
<tr>
<td>Ave</td>
<td>725</td>
<td>263</td>
<td>988</td>
<td>86</td>
<td>6.717x10^{-2}</td>
</tr>
</tbody>
</table>

**Table 8.2:** Fault-injected data for Traveling Salesman - 35 cities with \( t_{\text{seq}} = 3483 \) sec, \( t_{\text{orig}} = 531 \) sec, \( t_{\text{total}} = 988 \) sec and \( n=1 \):

\[
E_{\text{duplex}} = \frac{(3483/531)}{16 * 2} \times 100 = 20.5\% \\
E_{\text{ao}} = \frac{(3483/988)}{16} \times 100 = 22.0\%
\]

This suggests that some faults may not contribute an additional run time to the fault-tolerant algorithm. The types of faults which are of concern are the ones which directly affect the solution space. Since Branch and Bound is naturally redundant, in most instances, faults which appear in the solution and are not caught by the executable assertions will induce a delay, but will not affect the run-times of the final stage. It is only in the worst case of the silent worker that the final stage is affected. Experiments simulating the case of the silent worker were
performed. With the workers equally likely to be faulty, a fault simulating a silent worker was injected into each consecutively. The fault forced the worker to remain silent once its portion of the tree to expand was received. The results are shown in Table 8.2 for each worker.

Table 8.2 shows the decomposition of the fault-injected problem’s measured performance into the different components of Theorem 8.1. This decomposition allows the range of $t_{\text{delay}}$ to be computed for which the proposed approach is more efficient than the duplex system. A graph of the range for $t_{\text{delay}}$ is shown in Figure 8.1.

**Figure 8.1** A plot of efficiency comparisons for the Application-Oriented Fault-Tolerant and duplex system, varying the additional pruning factor $e$ and $t_{\text{delay}}$. Values for $t_{\text{delay}}$ and $e$ which fall within the shaded triangle on the upper left-hand side are candidates of the problem instance where the proposed approach is more efficient.

$$t_f = t_{\text{init}} + t_{\text{delay}} + (1-p+e) \cdot t_{\text{init}} \cdot n$$
$$t_f \text{ lineup} = [640 + t_{\text{delay}}] + (1-0.66+e) \cdot 640 \cdot n$$
$$t_{\text{delay}} < 206 \text{ secs}$$

In conclusion, a theoretical model was presented to show the different components involved in the fault-tolerant algorithm. This theoretical model was used as a basis for comparison with the duplex system. The theoretical model suggested that for certain values of $e$ and $t_{\text{delay}}$, the proposed approach would be more efficient than the duplex system. Experimental results validate the theoretical model.
IX. SUMMARY

This paper has shown how to apply Changeling to translate a verification proof into a fault-tolerant algorithm for the class of Branch and Bound problems. Executable assertions were shown to be related to the intermediate assertions of a verification proof and the consistency condition was relaxed due to natural redundancy in the program variables. The executable assertions were then applied to the program. A model was then developed to describe the behavior of the fault-tolerant algorithm. Experimental evidence validated this model and compared it to a duplex system which showed the range of values for which the Application-Oriented approach is better. Hence, the Application-Oriented Fault Tolerance approach proves to be efficient in terms of utilization of processors and in terms of cost, since this approach requires no explicit hardware or software redundancy.

Future research will examine liveness issues in the creation of fault-tolerant response systems.

REFERENCES


