05 Aug 1992

Formal Generation of Executable Assertions for Application-Oriented Fault Tolerance

Hanan Lutfiyya

Martina Schollmeyer

Bruce M. McMillin
Missouri University of Science and Technology, ff@mst.edu

FORMAL GENERATION OF EXECUTABLE ASSERTIONS FOR APPLICATION-ORIENTED FAULT TOLERANCE[†]

Hanan Lutfiyya, Martina Schollmeyer and Bruce McMillin

CSC-92-15

August 5, 1992

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.

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

Martina Schollmeyer and Bruce McMillin are with the Department of Computer Science at the University of Missouri-Rolla, Rolla, MO 65401.

Abstract: Executable assertions embedded into a distributed computing system can provide run-time assurance by ensuring that the program state, in the actual run-time environment, is consistent with the logical stage specified in the assertions; if not, then an error has occurred and a reliable communication of this diagnostic information is provided to the system such that reconfiguration and recovery can take place. Application-oriented fault tolerance is a method that provides fault detection using executable assertions based on the natural constraints of the application.

This paper focuses on giving application-oriented fault tolerance a theoretical foundation by providing a mathematical model for the generation of executable assertions which detect faults in the presence of arbitrary failures. The mathematical model of choice was axiomatic program verification. A method was developed that translates a concurrent verification proof outline into an error-detecting concurrent program. This paper shows the application of the developed method to several applications.

Index Terms - Formal Methods, Fault Tolerance, Program Verification, Concurrent Systems, Algorithms

I. INTRODUCTION

It is important for both life critical, and non-life-critical distributed systems to meet their specification at run time [LaLi92]. Large, complex, distributed systems, are subject to individual component failures which can cause system failure. Fault tolerance is an important technique to improve system reliability. The fault detection aspect identifies individual faulty components (processors) before they can affect, negatively, overall system reliability.

A failure occurs when the user observes that a resource does not perform as expected. The failure is the result of some part of the resource entering a state which is contrary to the specification of the part. The cause of the resource entering such a state is referred to as a fault. When a system can recover from a fault by either masking the fault and some how let with the system proceed without failure or by forcing failures to exhibit themselves then the system is said to have fault tolerance. Reliability is a measure of the probability that a specific resource will perform a required function for a specified period of time, usually the item's life time, even in the presence of faults. The higher the probability the higher the reliability of the system is considered to be.

Many methodologies for improving system reliability have been developed throughout the years. These different methodologies fall into two basic groups: fault masking techniques and concurrent techniques. Early attempts at improving system reliability used fault-masking methods; these methods make the hardware tolerant of faults through the multiplicity of processing resources. In contrast, concurrent fault detection methods attempt to locate component errors which can lead to system failure. Once the faults are identified, reconfiguration and recovery [YaHa84] are used to deal with the fault. This paper focuses on detecting the occurrence of errors. Recovery and reconfiguration are different issues. Work in concurrent detection methods include self-checking software [YaCh75] and recovery blocks [Rand75] which instrument the software with assertions on the program's state, watchdog processor [Aman78], which monitors intermediate data of a computation, and algorithm-based fault tolerance [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.

Application-oriented fault tolerance [McNi92], by contrast, provides a heuristic approach, based on the "Natural Constraints", to choosing executable assertions from the software specification. These executable assertions [YaCh75] in the form of source language statements are be inserted into a program for monitoring the run-time execution behavior of the program. The general form is as follows:

if $\neg$ ASSERTION **then** ERROR

Executable assertions are used to ensure that the program state, in the actual run-time environment, is consistent with the logical state specified in the assertion; if not, then an error has occurred and a reliable communication of this diagnostic information is provided to the system such that reconfiguration and recovery can take place. The heuristics for selection of the actual executable assertions are based on three metrics of *progress*, *feasibility*, and *consistency*.

What the earlier work lacks is a theoretical foundation built upon mathematical models and theories. In general, theoretical foundations can provide (1) criteria for evaluation, (2) means of comparison, (3) theoretical limits and capabilities (4) means of prediction, and (5) underlying rules, principles, and structure. This paper focuses on giving the application-oriented fault tolerance paradigm a mathematical model and then using the structure of the mathematical model to generate executable assertions which detect errors in the presense of arbitrary failures. The mathematical model of choice, in this paper, is axiomatic program verification.

This paper is organized as follows. In Section 2 we provide a brief overview of concurrent axiomatic proof systems which form the basis for our GAA system. Section 3 describes the HAA proof system for distributed programs and specifies the translation of a proof outline in this system to a error-detecting program through consistency. Section 4 presents several illustrative examples of the translation. In Section 5, we assess the concept of formal methods for generation of fault-tolerant programs.

II. A SHORT BACKGROUND ON PROGRAM VERIFICATION

The axiomatic approach to program verification is based on making assertions about the program variables before, during and after program execution. These assertions characterize specific properties of interest about 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 or sequence of statements 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]. P and Q are also referred to as a program *specification.*

In program verification logical assertions are determined that describe the effect of each statement that comprises program S. Each of these logical assertions must hold if the cumulative effect of these statements results in the execution of S satisfying the logical assertion Q. This collection of assertions is normally referred to as the "verification proof outline".

As a model axiomatic proof system, consider the work of [LeGr81], which is used for Hoare's model of concurrent programming, Communicating Sequential Processes (CSP) [Hoar78]. In this system, the first step is to prove appropriate properties about the individual processes in isolation. The individual statement axioms are omitted here except for the communication and parallel inference axioms.

The parallel inference rule is the following:

$$\frac{(\forall i\colon <P_i>\ S_i<Q_i>)\ \text{satisfied and interference} - \text{free}}{<(\forall i\colon P_i)>\ [\|_{i=1:n}\ A_i\colon S_i]<(\forall i\colon Q_i)>} \qquad \text{(parallel)}$$

The parallel rule (where A_i is the process label of the process that contains S_i) implies that construction of the proof of a parallel program can be derived from the partial correctness properties of the sequential programs it is comprised of. However, the sequential components of the parallel program have communication commands. The communication axiom is as follows:

$$<P>\ \alpha\ <Q> \qquad \text{(communication)}$$

where α is a communication command.

The communication axiom implies that full concurrent 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". Let us examine the proof outline of the matching communication pair [†].

$$\rho_1 :: [\cdots <P_1> \rho_2 ? x <Q_1>]$$
$$\rho_2 :: [\cdots <P_2> \rho_1 ! y <Q_2>]$$

The effect of these two communication commands is to assign y to x. This implies that $Q_1 \wedge Q_2$ is true after communication if and only if

$$(P_1 \wedge P_2) \rightarrow (Q_1 \wedge Q_2)^x_y \ddagger$$

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

The proof system in [LeGr81] (GAA proof system) 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". Any update of a global auxiliary variable on any processor is assumed to be immediately known to all processors.

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 [LuMc91] 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.

III. APPLICATION-ORIENTED FAULT TOLERANCE

Application-oriented fault tolerance works on the principle of testing at run time the intermediate logical assertions from the verification proof outline i.e. application-oriented fault tolerance works on the following principle:

If we test and ensure intermediate results of a program's computation meet its specification, the end solution meets its specification if the intermediate results meet their specification. If processor errors occur that do not affect the solution, then they are not errors of interest. Program verification provides these tests.

[†]The notation $\rho_i ? x$ in CSP notation denotes receiving a message into variable x from process ρ_i. Correspondingly, $\rho_j ! y$ denotes sending a message with value y to process ρ_j.

‡ This stands for the predicate $Q_1 \wedge Q_2$ with all instances of y replaced by x

The above principle yields a formal statement of application-oriented fault tolerance; we generate the executable assertions from the logical assertions used in the verification proof outline of <P>S<Q>. The executable assertion generated corresponding to any logical assertion Q_i from the verification proof outline is the following:

if $\neg Q_1$ **then** ERROR

Formally, this ensures that if P is true before the concurrent program S begins execution, S tests at run time that S satisfies the specification as defined by P and Q, by using the embedded executable assertions generated from the assertions of the verification proof. Conversely, the assertions of the verification proof represent the properties that must be satisfied by the run-time environment; an error that causes the execution of the program not to satisfy the specified assertions will be flagged as an error by the executable assertions.

The reader may be suspicious that some program S may be changed into a program S′ by a error that satisfies the specification as defined by P and Q. Consider, as an example, a program S computing some value x with postassertion $<Q> \equiv <x \geq 0>$. Suppose that S should compute $x = 3$. A program S′ may actually compute $x = 4$. The postcondition is still satisfied, although, the value is not what was intended. This is not a problem with the validity of the postassertion, it is a weakness of the specification. If $x = 3$ was what was really intended, then the proper postassertion should have been $<Q> \equiv <x = 3>$. If $<Q> \equiv <x \geq 0>$ is a sufficient specification for the application at hand, then there is no problem.

To eliminate confusion between the testing of intermediate results (via logical assertions) for correctness with respect to the algorithm and the evaluation of the executable assertions derived from the verification proof in the run-time environment we will refer to the former as the *verification environment* and the latter as the *(distributed) operational environment.*

To summarize, the transformation of an algorithm to an error-detecting algorithm involves using the assertions of the verification proof as executable assertions that are to be embedded into the algorithm. In order to accomplish this, however, we need to develop some additional theory: (A) a verification system that more closely models the distributed operational environment, (B) operational communication of state information through consistency, and (C) improving run-time efficiency.

3.A. History of Auxiliary Variable (HAA) Verification System

The logical assertions from the GAA verification environment cannot be directly used as executable assertions in the distributed environment; in the distributed environment, there are no global variables. Thus, to evaluate, at run time, logical assertions containing global auxiliary variables, an explicit updating mechanism must be created. Here we develop the verification proof system (HAA) in which updates of global auxiliary variables are exchanged at communication time. This matches, more closely, the operational environment. We show every verification proof outline in the GAA proof system has the same properties in the HAA proof system i.e. satisfaction and non-interference; thus, implying that the HAA proof system has the soundness and completeness properties of the original GAA proof system. The existence of the HAA proof system allows for proofs that can be directly transformed to executable assertions in the run-time environment.

Developing the HAA system requires us to keep track of which processes communicate with which other processes. Each process needs to record its global auxiliary variable updates with respect to all other processes. When communication occurs between two processes, they need to exchange the updates and locally apply the updates. This is formalized in the following definitions.

Definition 3.1: *For a process ρ_i,* h_i *denotes the sequence of all communications that process ρ_i has so far participated in as the receiving process. Thus,* h_i *is a list consisting of tuples (these are different from the* [Soun84] *tuples; all future reference to tuples will refer to the following tuples) representing matching communication pairs of the form*

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

where ρ is a process from which ρ_i receives from, Var is the variable that ρ is transmitting to ρ_i with formal parameter Val. T denotes the time at which the value Val was assigned to variable Var and C denotes the communication path. In Section B, we will use the path notion.

Since we have several processes running in parallel and there exists no concept of a global time, the time T is a local time represented by an 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 3.2: *Let* g_{ij} *depict the GAV set in process ρ_i with respect to process ρ_j, i.e.,* g_{ij} *contains the changes that were made to the GAVs in ρ_i since the last communication with ρ_j.* G_i *is the set of sets* $g_{i0}, g_{i1}, \ldots, g_{i(N-1)}$ *in process ρ_i. Thus, when two processes ρ_i and ρ_j communicate, the respective subsets,* $g_{ij} \in G_i$ *and* $g_{ji} \in G_j$, *are exchanged.*

When two processes ρ_i and ρ_j communicate, where ρ_j is the sender, ρ_j will augment the communication by sending the values of global auxiliary variables that ρ_j updated or received updates of between the last and current communications between ρ_i and ρ_j. We batch the changes made to the local copies of the global auxiliary variables by p_j since the last communication (with any other processor) in g_{jj}. Before a communication, the function ψ applies changes to all g_{jk}'s and g_{jj} is reset to null to collect future changes. Definition 3.4 formally describes the communication of g_{ji}. Definition 3.5 formally describes how process ρ_i updates G_i based on the communicated g_{ji} after communication has taken place.

Definition 3.3: *The actual set of GAVs to be sent during a communication between ρ_i and ρ_j, where ρ_j is the sender, is determined based on the variables in* g_{jj}, *i.e., all the variables that were updated in ρ_j since the last communication with any process. The set* g_{jj} *is updated every time an assignment to a GAV takes place in ρ_j and reset at communication time to the empty set. The following function* $\psi(G_j, g_{jj})$ *describes the update of all variable histories before a communication.*

$$\rho_j\text{:}\quad (\forall k, 0 \le k \le N-1)(\forall g_{jk} \in G_j)$$
$$[\text{ if } k \ne j \text{ then } g_{jk}^{T+t_j} \leftarrow (g_{jk}^{T} - g_{jj}^{T}) \cup g_{jj}^{T+t_j} \text{ else if } k = j \text{ then } g_{jj}^{T+t_j} \leftarrow \varnothing \ ]$$

where T represents the local time any last communication and $T + t_j$ *represents the local time of the current communication.*

The following definition formally defines the semantics of global auxiliary variable communication.

Definition 3.4: *The primary communication is a matching communication pair for the exchange of variables between processes which are not GAVs. It can be described by a tuple* $[\rho_j, (Var, Val), T_i = t, j]$ *where t is the current value of the local time. It is easy to see that all communications in the GAA system are primary since GAVs are updated globally. An augmented communication permits the exchange of the GAVs after a primary communication occurs.*

For each process ρ_i after an augmented exchange with ρ_j, ρ_i updates its set of GAVs in G_i with the new values received. This interchange is described in Figure 3.1 for two processes ρ_i, ρ_j and one matching communication pair within the execution sequence of the two processes.

Definition 3.5: *The updates performed in the different processes are described by a function* $\phi(G_i, g_{ji})$ *on the set of the GAV history and the variables to be updated. The actual update function* ϕ *is now defined on all the subsets within* G_i *on tuples of the form* $[\rho_j, g_{ji}, gvar_j^{T+t_i}, T + t_i, j]$.

$$\rho_i\text{:}\quad (\forall k, 0 \le k \le N-1)(\forall g_{ik} \in G_i)$$
$$[\text{ if } k \ne j \text{ then } g_{ik}^{T+t_i} \leftarrow (g_{ik}^{T} - gvar_j^{T}) \cup gvar_j^{T+t_i} \text{ else if } k = j \text{ then } g_{ij}^{T+t_i} \leftarrow gvar_j^{T+t_i} \ ]$$

where T represents the local time of the last communication and $T + t_i$ *the local time of the current communication.*

When processes ρ_i and ρ_j communicate, all old values in the set g_{ij} will be replaced by the new variables. This operation removes all elements that the two sets have in common and then combines them with the updated GAVs. The only exception will be the set g_{ij} when processes ρ_i and ρ_j are communicating. In that case all old variables need to be removed and be replaced by the new values.

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 will have different values within their set of GAVs. We show that because of non-interference, this is not a problem with respect to the proof system.

Within a process execution, two communicating processes can have arbitrary interleavings of their statements up to the communication, but are conceptually synchronized at the communication point. The assertions will not interfere with each other due to the non-interference property of the GAA system which provides for arbitrary execution orders. Since two (or more) processes will only change (write onto) the same global auxiliary variable if they have to communicate with each other, they will also exchange other variables/data in that process and the values of the auxiliary variables will be available for the other process at the critical point: right after a communication takes place. Thus, sending only the

```
For process $P_i$:
/* execute arbitrary set of statements excluding communication but including assignments to auxiliary variables */
$S_{i1}$; $<T_i:=1>$
$S_{i2}$; $<T_i:=T_i + 1>$
 . . . ;
$S_{ik}$; $<T_i:=T_i + 1\}$
/* update the auxiliary variables */
$G_i \leftarrow \psi(G_i, g_{ii})$; $<T_i:=T_i + 1>$
/* perform communication with process $P_j$; the first communication represents the actual communication */
/* the next two communications represent the exchange augment of the auxiliary variables */
$P_j$ ? V ; $<T_i:=k>$
$P_j$ ? $g_{ji}$ ; $<T_i:=k+1>$
$P_j$ ! $g_{ij}$ ; $<T_i:=k+2>$
/* update the auxiliary variables */
$G_i \leftarrow \phi(G_i, g_{ji})$; $<T_i:=k+3>$

For process $P_j$:
/* execute arbitrary set of statements excluding communication but including assignments to auxiliary variables */
$S_{j1}$; $<T_j:=1>$
$S_{j2}$; $<T_j:=T_j + 1>$
 . . . ;
$S_{jk}$; $<T_j:=T_j + 1>$
/* update the auxiliary variables */
$G_j \leftarrow \psi(G_j, g_{jj})$; $<T_j:=T_j + 1>$
/* perform communication with process $P_i$; the first communication represents the actual communication */
/* the next two communications represent the exchange augment of the auxiliary variables */
$P_i$ ! V ; $<T_j:=k>$
$P_i$ ! $g_{ji}$ ; $<T_j:=k+1>$
$P_i$ ? $g_{ij}$ ; $<T_j:=k+2>$
/* update the auxiliary variables */
$G_j \leftarrow \phi(G_j, g_{ij})$; $<T_j:=k+3>$
```

Figure 3.1. An HAA proof outline for one matching communication pair.

history of the global variable updates instead of immediately providing the other process(es) with the latest information will not cause any problems, since the values of the variables will be available at the communication points, where they are in fact provided.

An example of three possible process execution sequences that are subject to non-interference are shown in Figure 3.2. For any two processes, non-interference will guarantee that the execution order of the two processes or any arbitrary interleaving of them will not invalidate the assertions made on the respective process statements.

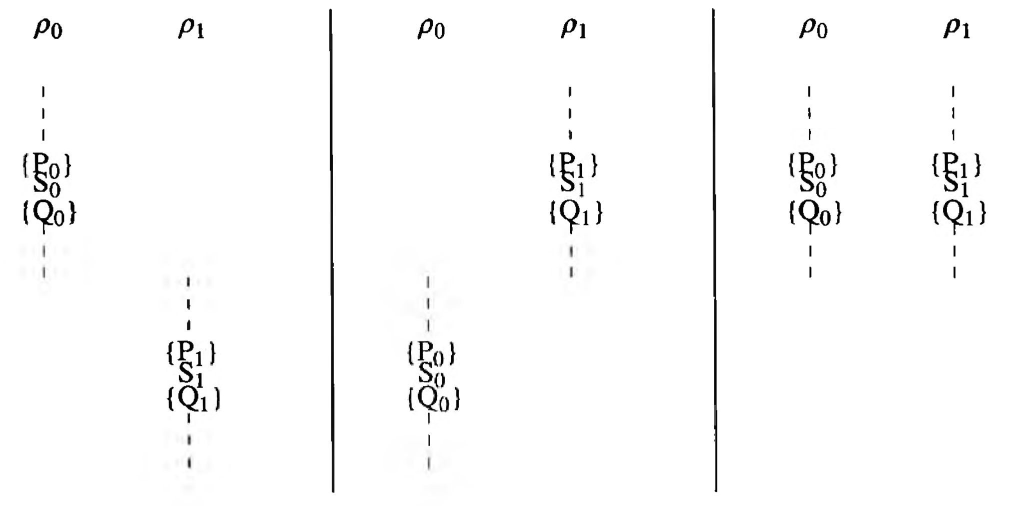

Figure 3.2. Some possible process execution sequences before communication takes place.

The proofs of Appendix A formally show how non-interference and the rule of satisfaction can be used to show that the soundness and completeness properties of the original GAA system will hold in the new HAA proof system.

Theorem 3.1: *The history of auxiliary variables approach (HAA) retains the properties of the global auxiliary variables approach (GAA).*

Proof: Theorems A.1, A.2, A.3, and A.4 show that the conversion from GAA to HAA preserves the inference rules as well as soundness and completeness based on the preservation of non-interference and satisfaction. □

3.B. 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, However, we are concerned with the distributed faulty environment. Thus, it is necessary to ensure that faulty processors cannot fool executable assertions by incorrect augmented communication of g's through sending inconsistent messages to different processors. It is necessary for this to be detected. This is the purpose of *consistency* executable assertions. Mathematically, this can be described as follows:

Definition 3.6: *For a non-faulty process p_i, if there* exists any two tuples t_1, $t_2 \in h_i$ *such that*

$$t_1 = [j, (Var, Val_1), T, C_1]$$

$$t_2 = [j, (Var, Val_2), T, C_2]$$

then if $Val_1 \circ Val_2$ *the system is said to be inconsistent otherwise the system is said to be consistent.* $\circ$ is defined as a set of functions such that each $o' \in o$ is of functionality dt $\rightarrow \{T, F\}$ where dt is an abstract data type. Examples of o' are $\neq$, $\subseteq$, $\neg$prefix, or some other operator appropriate to the choice of the data type of Var. Where no ambiguity results, we will refer to a particular o' simply as o.

The strongest motivation for the consistency condition is to supplement the power of the executable assertions derived from the HAA system. When the value of a variable computed in 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, the consistency definition of 3.6 ensures that a non-faulty processor receives a consistent set of input values for its executable assertions, otherwise, $Val_1 \, o Val_2$, and an inconsistent system can be detected. The degree of fault tolerance is based on standard network flow arguments and is not repeated here. It should be noted that all faults in communication links are mapped to a processor, thus it is enough to assume only faulty processors.

Consistency does not have to be explicit. In other words, an error-detecting program may have to explicitly add code to implement consistency. This can be done in many ways. 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 is state i, the 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. Natural redundancy will be used when we consider the Branch and Bound example of Section IV.

3.C. 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 run-time efficiency is considered, 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 for that reason.

3.D. Section Summary

This section showed how the properties of the GAA system are preserved when a conversion from this formal system into another occurs; non-interference is the critical point that assures that satisfaction, soundness and completeness can be retained in the new system. The new system is important since it simplifies the operations on sets of executable assertions which are used to provide fault tolerance in a distributed operational environment. We then showed how the HAA system can be transformed, through consistency, into an operational system and made some comments on run-time efficiency.

We have applied this transformation to several concurrent applications ranging from concurrent database transactions schedules [LuSM92a], concurrent sorting [LuSM92b] and concurrent branch and bound [LuSM92c]. We have obtained performance and error coverage data on each. The next section presents example applications of this system including its verification, transformation, measures of its efficiency, and error coverage.

4. EXPERIMENTAL RESULTS

The transformation on an application to an error-detecting code is straightforward; however, there are numerous applications in which various heuristics improve run-time efficiency. The applications selected for this section reflect this diversity in the use of the transformation.

4.A. Distributed Database Concurrency Control

Distributed database applications are a wide use of distributed systems that can be implemented on a network of workstations. Fundamentally, processes execute *transactions* which perform *Lock*, *Read*, *Write*, and *Unlock* operations on *entities* stored in the database [BeGo82]. The interleaving of operations from the various transactions, or *schedule* of operations, performed concurrently must be equivalent to the effects of executing the transactions serially [EGLT76]. This is most commonly achieved by the *Two-Phase Locking* protocol (transactions finish locking entities before unlocking any). In the system model, m transaction manager (TM) processes execute transactions which access data uniquely held by n data manager processes each controlled by lock manager (LM) processes. This example creates a program which detects faults in the scheduling part of the lock manager.

The verification of the two-phase locking protocol in the GAA system is straightforward (the interested reader is referred to Appendix B.1). The assertion of interest is an invariant, I, that expresses that the transactions are well-formed† and that transactions do not simultaneously modify the same entity. This requires assertions on two auxiliary variables: schedule_i, where $0 \le i-1 \le N-1$, denotes the schedule of operations as ordered by the lock manager on process i and SG denotes the partial order of dependencies on the access of the shared entities by the transactions.

In conversion from the verification proof in GAA to HAA, g_{ij} for any two processors i and j is

$$g_{ij} = \{\{\text{schedule}_r \mid 0 \le r \le N-1\}, SG\}$$

Since coordinating transaction managers will request use of other entities managed by other lock managers, then any lock manager i may receive values of schedule_j, where $i \ne j$. Consistency is checked by showing that the last update of schedule_r, where $0 \le r \le n-1$, coming from TM_s is a prefix of the last update of schedule_r coming from TM_t or vice-versa. Thus, the $\circ$ operator for consistency is simply the prefix operation on two schedules: $\text{schedule}_r \in g_{si}$ and $\text{schedule}_r \in g_{ti}$ of lengths L_1 and L_2, respectively and $L_1 \le L_2$ is defined as follows:

$$\circ \equiv \neg(\exists k(0 \le k \le L_1 - 1 \wedge \text{select}(k, \text{schedule}_r \in g_{si}) \ne \text{select}(k, \text{schedule}_r \in g_{ti})))$$

The function select is used for determining the kth element of the schedule sequence.

† A well-formed transaction does not attempt to lock an entity that has already been locked nor does it read or modify an entity unless it has already locked that entity.

The transaction managers are being used to test the executable assertions. By the noninterference property of the HAA system, the transaction managers only test available data; no extra communication is forced to obtain a global picture of the entire system. The lock managers use consistency to ensure that the other lock managers are correctly sending g_{ij}'s to a particular transaction manager.

Since we evaluate the invariant, I, at run-time in all processors, we can detect any schedule that violates I under a consistent system. Thus, we need to derive bounds on the number of faulty processors such that consistency is maintained.

Analysis of the expected error coverage is important for any error-detecting algorithm. It is assumed that a transaction manager uniformly requests access to data at all sites. This implies that a transaction can communicate with all lock managers. We have shown [LuSM92a] that if we assume n lock managers and m transaction managers that the system can handle n-1 faults occurring on the processors with the lock managers if no transaction managers fail or the system can tolerate m-1 faults occurring on processors with the transaction managers if no lock managers fail.

From the construction of g_{ij} by the functions ψ and ϕ the actual set of auxiliary variables sent during a communication between LM_i and TM_j is the differences in $schedule_r$, where $0 \le r \le n-1$, and SG sent in the previous communication between LM_i and TM_j and the current communication. Since a transaction manager uniformly requests access to data at all sites, then the average difference between old and new schedules is m entries.

SG is updated when there are two transactions that request access to a common entity. In the worst case, there are m entities in common. Therefore, SG increases by m elements. In the basic algorithm there is one action communicated between LM_i and TM_j. Let this message length be denoted by B. In the transformed algorithm the average message length is (nm+m+1)B giving an overhead of (nm+m+1) = O(nm).

On the surface, this may seem like an a great deal of overhead. However, the major part of the cost of communication is in the connection set up time between the two communicating processes and not in the actual transfer. Therefore, the additional cost of piggybacking auxiliary variables is minimal. This can be better understood by modeling the communication explicitly. The time to transfer data between two processes is S + RL where S is the setup time, R is the transfer rate (in seconds/byte) and L is the length of the message. Typical numbers for these values (taken from a Sun 4/20 using TCP/IP on an IEEE CSMA/CD) yield, S =16 msec and R is $(10\text{Mb/sec})^{-1}$. The additional message length will impact the message transfer time when $RL \ge S$. For this model system, $L \ge 10^4$ and for a 10 byte message, (B = 10), $nm \ge 1000$.

The derived executable assertions can be implemented in linear time on the length of schedule and the size of SG. However, as time progresses, schedules from the lock managers will continue to grow. However, it is clear that at some point, old schedule information is no longer necessary. Thus, the GAV set for $schedule_i$ can be reduced by deleting this old information.

4.B. Parallel Sorting

Parallel implementation of sorting is common in algorithmic study. Of particular interest is the bitonic sort on the hypercube [Quin87] due to its nice recursive definition. Bitonic sort's fundamental data structure is a bitonic sequence which consists of a subsequence of ascending elements followed by a subsequence of descending elements, or vice-versa. The basic operation is to "sort" bitonic sequences, in parallel, into ascending or descending sequences, which, when concatenated together, form a longer new bitonic sequence. This procedure is recursively performed until only an ascending or descending sequence of elements remains. The parallel time complexity of bitonic sort is $O(\log_2^2 N)$ where N is the number of processors (data elements).

The naturally expected result at termination of a sorted permutation of the original input is shown by verification (Appendix B.2). Intuitively, we must show, that at each level of recursion, (1) bitonic sequences are maintained, (2) each sorted sequence is a permutation of the previous bitonic sequences. Two global auxiliary variables; ia_l and pa_l relate the current and past values of a_l, the local copy of processor l's value to be sorted. In the loop formulation (Appendix B.2), two loop invariants result from the verification; the outer loop shows properties (1) and (2) directly while the inner loop assertion supports the loop's contribution to the outer loop.

In conversion from the verification proof in GAA to HAA, g_{ij} for any two processors k and l is

$$g_{kl} = \{\{ia_m\}, \{pa_m\} \mid 0 \le m \le N-1\}$$

We now turn our attention to observations on efficiency of the transformed program. We can apply heuristics to reduce the overhead penalty of the error-detecting algorithm.

Even after deleting all assertions that use only local variables, we are left with redundant assertions. Consider the inner and outer loop invariants; each of these use global auxiliary variables. If the inner assertion fails, then the outer assertion also fails. Thus, for efficiency, we can delete the inner assertion from the transformed program.

The auxiliary variable pa_l does not need to be communicated directly within the augmented communication. At the end of each iteration of the outer loop, we may simply assign $pa_l \leftarrow ia_l$ since the current bitonic sequence values become the old bitonic sequence values in the next iteration. Thus, we can modify g, as follows

$$g'_{kl} = \{\{ia_m\}, \mid 0 \le m \le N-1\}$$

For consistency, since we have a regular, point-to-point interconnection graph, the natural communication patterns define when consistency can be tested. Formally,

$$\circ \equiv ia_l \in g_{kl} \ne ia_l \in g_{ml};\ l \in \lambda^l_{ij}$$

where λ^l_{ij} (definition given in Appendix B.2) reflects the natural flow of augmented communications that processor l receives at time i, j where i and j are values of the outer and inner loop indices, respectively. It is shown in [LuSM92b] that for each processor k in λ^l_{i0} and $i > 0$ that processor l receives two values of a_k in stage i that were computed in stage $i - 1$.

The parallel bitonic sort algorithm of this section, has a time and communication complexity of $O(\log_2^2 N)$. Implementing the assertions from HAA with the efficiency improvements described above yields an error-detecting algorithm time complexity of O(N) computation and $O(\log_2^2 N + N \log_2 N)$ communication tolerant of one faulty processor [McNi92]. Implementation shows, that as system size increases, the performance penalty lessens.

4.C. Branch and Bound

Our last example exhibits two important properties, consistency through natural redundancy, and loose synchronization. In a parallel implementation of branch and bound, N individual processes search portions of a state space tree for the lowest cost node. As the search progresses, processes propose improving bounds on the solution value which allows all processes to prune their search space.

Verification in GAA of a parallel branch and bound requires showing the postcondition, Q: the entire search space is eventually considered, either by direct exploration, or by pruning and that the lowest cost solution (GAV $solution_i$ in process i), out of this exploration, is the one reported. Details are given in Appendix B.3. The conversion to the HAA proof system is straightforward. Transformation to an error-detecting algorithm, however, exhibits some interesting phenomena.

Explicit consistency, in the sense of Section III.B, can be implemented. However, Branch and Bound algorithms are *Naturally Redundant* [LaMG91]. Specifically, this implies the effects of a faulty process proposing an erroneous bound will be naturally corrected by the algorithm at some later time of the computation; if the erroneous bound is larger than it should be, not as many solutions will be pruned by other processors, but in the end, the postcondition will still be met. Appendix B.3 gives a formal proof and further analysis. Natural redundancy takes place of consistency in most cases; however, since the processes involved in the solution are not tightly synchronized, the absence of a message cannot be detected; other techniques are needed.

Since message absence cannot be detected, it is possible, in the faulty environment, that the optimal solution may be pruned off or that the subtree that the optimal solution is in is not communicated properly due to a faulty processor withholding information. However, since absence of information does not constitute an error in executable assertions derived from HAA, the error-detecting program's executable assertion corresponding to the postassertion Q in HAA, is satisfied, but the program has not actually met the postcondition. A brute-force way of testing the postcondition is for each process to engage in N distributed agreements [LaSP82] on $solution_i$ by broadcasting its perception of the best solution to all other processes. However, this is more then we need. Instead a series of validation rounds are used to check the validity of the solution.

Each validation round consists of verifying Q by redistributing the initial states to different workers and restricting the workers to communicate within disjoint sets of workers. If the validation round finds a discrepancy in $solution_i$, then there is an error. In terms of efficiency, since a solution has already been found that has a bound lower than most paths in the tree, many branches can be immediately pruned. To counter the effects of more than one faulty processor, additional validation rounds are necessary. For full details of an error-tolerant algorithm using validation rounds see [SuMc91].

With a low cost as an upper bound, the verification stage only requires very little time to verify the solution. If the bound is quite large, the performance for the verification stage could be bad, but no worse than the initial search round. Experimental results [SuMc91] show that the overhead of the validation round is less than 50 percent of the time to compute the original solution.

V. ASSESSMENT of the METHOD

In the beginning of this paper we argued that previous work on generation and selection of actual executable assertions for concurrent programs lacks a sound, theoretical basis. We have developed a fundamental technique which translates a concurrent verification proof outline into a error-detecting concurrent program on a sound, theoretical basis. In this section we assess the method as a basis for building fault-tolerant concurrent programs.

5.A. Translator

To prove the concept of formal methods for application-oriented fault tolerance requires a large number of programs to be treated. Currently to test an application within the fault tolerance framework requires execution of the translation algorithm by hand. The automated translation from a proof outline in the GAA proof system specified using CSP to the HAA proof system is straightforward.

There are two issues involved in the generation of consistency executable assertions from the HAA proof system: the determination of the $\circ$ operator for the abstract data type is not defined the the abstract data type itself. For example, in the database application the $\circ$ operator for the sequence was defined as a prefix operation. In other applications, the $\circ$ operator is defined as an equivalence operation.

The insertion of code for consistency into the error-detecting code is the job of the translator. Syntactically, consistency is checked after each augmented communication. Should it be the job of the translator to insert code only where consistency can be checked? The answer is no. The definition of consistency states that "when two or more tuples are available, the consistency operator is applied." Two or more tuples may not be present at every augmented communication. Thus, a consistency test at these places is unnecessary and adds to overhead. This is a run-time issue at best, and an efficiency issue at worst.

5.B. Assessing Fault Coverage

How is fault coverage measured? There are several ad-hoc approaches to measuring the fault coverage of an instrumented system based on the application. We turn to formal methods to analyze error coverage. Since the run-time error-detecting program created by the transformation process forms a logical system, it is possible to reason about this system using automated theorem proving techniques. The difficulty is in determining the notion of an error. Roman in [Roma87] describes an augmented CSP called CSPS which takes into account details of the underlying machine architecture. Errors are then simulated in the programming system through the nondeterministic selection of faulty paths through the system. Program verification can then be applied to this augmented program to show that fault tolerance properties hold at run time.

5.C. Efficiency

How do we determine the minimal set of executable assertions. As we have seen with the parallel sort it is not necessary to use all verification proof assertions as executable assertions. We saw that that some assertions have the same coverage as others or ,in other words, some assertions are *subsumed* by others. We are currently developing an algorithmic way of determining which assertions are subsumed.

5.D. Liveliness

We have made use of axiomatic proof systems for CSP-like programs in our current work. We have been able to construct run-time error-tolerant programs. It is clear, however, that for concurrent, real-time systems, we need a general way of reasoning about eventuality, particularly to show total correctness. A temporal logic-based proof system [OwLa82] provides a convenient, expressive, way of reasoning in time. The problem of relating temporal assertions to executable assertions, however, is not straightforward, as in the axiomatic techniques. The complexity of the general problem of relating temporal assertions to those in first order logic is disheartening. Thus, the need to explore further the notion of temporal proof systems.

5.E. Fault-Detection

Little of our work so far has been concerned with locating a fault when an error occurs. We have concentrated mainly on identifying failures while ignoring the concept of reconfiguration and recovery. To perform reconfiguration requires isolation of thc faulty component [YaHa84]. Future research will approach this problem based on the formal methods paradigm outlined in Reiter's theory of diagnosis from first principles [Reit87].

5.F. Reconfiguration and Recovery

It is possible that reconfiguration can be done in a local distributed manner. Indeed some preliminary work has attempted application oriented reconfiguration for relaxation techniques [OsKA86] and the reconfiguration on rings embedded in a faulty hypercube [LiMc92]. Application oriented reconfiguration involves not replacing the hardware component, but remapping portions of the application to fault-free components.

APPENDIX A. PROOFS OF SOUNDNESS AND COMPLETNESS

A.1. Non-Interference. Non-interference of a set of parallel processes is defined such that for each assertion P in a process ρ it must be shown that P is invariant over any parallel execution. A command S is *parallel to* an assertion P if S is contained in a process of a parallel command and assertion P is contained in a different process of the same parallel command. Now every command S parallel to P must satisfy

$$<P \wedge pre(S)> S <P>$$

and every matching communication pair $S \equiv \rho_1!y$ and $R \equiv \rho_2?x$ that is parallel to P, must satisfy the condition

$$(P \wedge pre(S) \wedge pre(R)) \Rightarrow P^x_y$$

A proof of non-interference is mechanical by comparing every assertion in every process against every command in every other process and against every matching communication pair.

In the HAA system, the preconditions of the matching communication pairs are derived from the preconditions in the GAA system and are therefore implied by them. Thus, $pre(S)_{GAA}$ will imply $pre(S)_{HAA}$ for an arbitrary statement S. P_{GAA} and P_{HAA} will correspond to the same assertion in both the HAA and the GAA system.

We now need to show that for an assertion P and a statement S involving the set of GAVs in the GAA system, if $<P_{GAA} \wedge pre(S)_{GAA}> S <P_{GAA}>$ is true then the corresponding expression in HAA will also be true. Let's assume that this is not true, i.e., the statement is true in GAA but not in HAA, then there must exist some assignment of the GAVs in HAA such that $<P_{HAA} \wedge pre(S)_{HAA}> S <\neg P_{HAA}>$. This can only happen if S invalidates P.

Lemma A.1: *The proofs are interference-free if S is not a communication command.*

Proof: Assume they are not interference-free. Then there exists some S in ρ_i such that S makes some assignment x: = y which, at some future time, is communicated.

Case 1: S in ρ_i does not involve variables in G_i. Therefore, since S is local to ρ_i, it cannot affect the truth or falsity of assertion P_{HAA} in process ρ_j.

Case 2: S is an assignment to the GAVs. Eventually, the change x: = y, where $x \in G_i$, that was performed by S will be communicated to ρ_j. If this invalidates P_{HAA} then this is equivalent to $<P_{GAA} \wedge pre(S)_{GAA}>$ S $<\neg P_{GAA}>$, which by non-interference in GAA cannot happen.

Cases 1 and 2 show that the proofs cannot interfere if S is not a communication command. □

Lemma A.2: *The proofs are interference-free if S is a communication command and R and S form a matching communication pair.*

Proof: It needs to be shown that if

$$(P_{GAA} \wedge pre(S)_{GAA} \wedge pre(R)_{GAA}) \Rightarrow P^x_y\ _{GAA}$$

is true then also

$$(P_{HAA} \wedge pre(S)_{HAA} \wedge pre(R)_{HAA}) \Rightarrow P^x_y\ _{HAA}$$

must be true. We know that the preconditions of GAA imply the preconditions of HAA. Also, we know that P_{GAA} and P_{HAA} correspond to the precondition to the same communication. After the communication and the variable assignment x: = y, $P^x_y\ _{GAA}$ still corresponds to $P^x_y\ _{HAA}$ since the same assignment of variables was performed in both systems and since by case 2 of Lemma A.1, any communication of GAVs will not invalidate any assertions in HAA. Now, if the GAA system is interference-free but the HAA system is not, then this will lead to a contradiction. □

Lemmas A.1 and A.2 showed that the HAA system is interference-free if the GAA system is interference-free. This is summarized in Theorem A.1.

Theorem A.1: *Non-interference of the GAA proof system implies non-interference of the HAA proof system.*

Proof: In the GAA proof system every program that can be verified using this approach requires non-interference of the GAVs. Thus, every program that can be verified using the GAA system is composed of interference-free proofs, and from Lemmas A.1 and A.2 we can immediately conclude the non-interference of the HAA proof system. □

A.2. Satisfaction Proof. For the satisfaction proof we again need to consider matching communication pairs of the form

$$S \equiv <P_1>\rho_1!\ y<Q_1> \text{ and } R \equiv <P_2>\rho_2?\ x<Q_2>$$

When processes ρ_1 and ρ_2 communicate, an assignment x: = y is performed. Therefore, $(Q_1 \wedge Q_2)$ is true after the communication if and only if $(Q_1 \wedge Q_2)^x_y$ is true before the communication takes place. Before the communication, however, both preconditions of the matching communication pair must also be true, or else no communication will take place. Thus, we can see that $(P_1 \wedge P_2)$ must be true. The rule of satisfaction now requires that every matching communication pair must satisfy

$$(P_1 \wedge P_2) \Rightarrow (Q_1 \wedge Q_2)_y^x$$

For the HAA system we perform the satisfaction proof only after the update of the GAVs by ϕ (after the communication), whereas in the GAA system the GAVs will always be consistent and we do not have to wait for their update.

Theorem A.2: *The satisfaction proof of GAA implies the satisfaction proof of HAA.*

Proof: Since the assertions in the HAA system are derived from the assertions in the GAA system, the preconditions in GAA for the matching communication pairs imply the preconditions in HAA. This is denoted by

$$(P_{1,GAA} \wedge P_{2,GAA}) \Rightarrow (P_{1,HAA} \wedge P_{2,HAA})$$

Both proof systems have the same matching communication pairs and since updates of the global variables are performed in the HAA system after each communication and due to Theorem A.1's statement of non-interference, the postconditions after the GAV update in the HAA system correspond to the ones in the GAA system. Therefore the following statement must be true as well:

$$(Q_{1,GAA} \wedge Q_{2,GAA})_y^x \Rightarrow (Q_{1,HAA} \wedge Q_{2,HAA})_y^x$$

The GAVs in the HAA system may be inconsistent in the different parallel processes but, again, due to Theorem A.1, this is not of concern. We now need to show

$$(P_{1,HAA} \wedge P_{2,HAA}) \Rightarrow (Q_{1,HAA} \wedge Q_{2,HAA})_y^x$$

to prove that HAA can be satisfied if GAA is satisfied.

Let us assume now that $(Q_{1,HAA} \wedge Q_{2,HAA})_y^x$ is false but the preconditions $(P_{1,HAA} \wedge P_{2,HAA})$ are true. From the expressions above we know that if the preconditions in the GAA proof system are satisfied, then the preconditions in the HAA system must also be satisfied since they are derived from the GAA system. Satisfaction of the preconditions and successful communication in the GAA system imply that the postconditions in the GAA system are also satisfied. This, however, implies that $(Q_{1,HAA} \wedge Q_{2,HAA})_y^x$ must also be true, according to the statement above. Now, if the postconditions in the HAA system are not true, which is the assumption stated above, this will lead to a contradiction. Thus, the satisfaction proof of GAA implies the satisfaction of HAA. □

A.3. Soundness. We need to show that soundness is preserved in the conversion from one formal system into another. This means that we can show that with the rules given in a proof system, all expressions that can be derived by it are logically implied.

The following theorem shows how the conversion from GAA to HAA preserves soundness.

Theorem A.3: *If the GAA proof system is sound then the HAA proof system is also sound.*

Proof: It can be seen that in the conversion described in this paper (from GAA to HAA) the inference rules are not changed and the only changes made affect the communication between processes and the updates of the GAVs. This is shown by the satisfaction proof which implies that since the expressions in the GAA system could be shown to be satisfiable that now the expressions in HAA are also satisfiable. Thus, all statements that could be derived in the GAA system can now be derived in the HAA system. Since we assume GAA to be sound, only statements that are logically implied could be derived. The same property must now also hold for HAA. □

A.4. Completeness. It is not always possible to show that a formal system is complete, i.e., that every statement that is logically implied can be derived by this system. If we assume that the GAA system is complete (in a restricted sense) [Owic75], we would like to show that the transformation to HAA is also complete.

Theorem A.4: *If the GAA proof system is complete (in a restricted sense) then the HAA proof system is also complete (in a restricted sense).*

Proof: The set of assertions in the GAA system are "copied" in the conversion to the HAA system. Each assertion that was originally present in a proof of the GAA proof system will have an equivalent assertion in the HAA system. This can be described by a one-to-one function. Now, if a new assertion is introduced in the HAA system, then it must be logically implied by the given statements. This means that a corresponding assertion can be introduced in the GAA system. This function is the identity on the set of assertions in the HAA proof, which is also a GAA proof. Thus, any assertion in GAA will also be in HAA, and any assertion introduced in the HAA system is an assertion in the GAA system. Therefore, from Theorems A.1 and A.2, if the GAA system is complete, so must be the HAA system. □

A formal system should be sound and complete. The formal system introduced by [OwGr76] is shown to be complete in a restricted sense in [Owic75]. The GAA system that is used here as a transformation basis for the HAA system is related to the system introduced by [OwGr76]. However, soundness or completeness are not formally shown. Our results are no stronger than the existing results on the soundness and completeness of the GAA system.

APPENDIX B. VERIFICATION PROOF OUTLINES for EXAMPLES

B.1. Verification and Translation Of a Lock Manager Process

The highlights of the verification proof in the GAA system are presented here. The following auxiliary variables are used in the verification proof: schedule_i and SG. schedule_i denotes the schedule of operations as ordered by the lock manager on process i. SG denotes the partial order between transactions.

A transaction T_{kl} finishes before a transaction T_{st}, if all the operations in T_{kl} occur before all the operations in T_{st}. We can define a relation "<" on the set of transactions as the smallest relation satisfying the following condition: If T_{st} requests a lock on entity e that has been previously locked by T_{kl} then $T_{kl} < T_{st}$. Since, T_{st} must wait for T_{kl} to release the lock and since, locks are not released until the termination of the transaction, then it follows that T_{kl} finishes all its operations before T_{st} finishes all its operations.

It does not make sense for T_{kl} to finish before T_{st} and for T_{st} to finish before T_{kl}. Therefore, "<" is antisymmetric. If T_{kl} finishes before T_{st} and T_{st} finishes before T_{yz} then T_{kl} finishes before T_{yz}. Therefore, "<" is transitive. Also, since, it does not make sense to say that a transaction finishes before itself. This implies that the relation "<" is irreflexive. Since, "<" is transitive, antisymmetric and irreflexive then "<" is an irreflexive partial order. If two transactions do not operate on common entities then it is impossible to say which transaction completes before the other. Therefore, "<" is not a total order.

The partial order is denoted by a graph represented by SG = (T, E), where T is the set of transactions, and the set E represent the arcs between the transactions. An arc between T_{kl} and T_{st} implies $T_{kl} < T_{st}$. Since, SG is a graphical representation of a partial order, then SG does not contain cycles, i.e. it is acyclic. SG represents a partial order of transaction execution. This order of transaction execution represents a dependency among transactions that indicates which transactions must be completed before others. The partial order represented by SG can be shown to be equivalent to the partial order defined in [EGLT76], which shows that if each set of transactions is well-formed, two-phase, then any legal schedule for T is consistent and that the partial order can be extended to a total order that defines a serial schedule that is equivalent to the run-time schedule.

The following functions on schedule_i are used in the verification proof:

$\text{Last\_Op}_i(e, j)$: This represents the jth to the last operation on entity e.

$\text{Last\_Trans}_i(e, j)$: This represents the jth to the last transaction to operate on entity e.

The precondition for each lock manager process running on processor i assumes that no scheduling of any of the transactions has taken place. At termination of the lock manager on processor i, we want SG to denote an irreflexive partial order, i.e. SG must be acyclic and that the schedule produced by a lock manager on process i is legal.

The best way of ensuring that SG is acyclic at the end of termination of the lock managers is to ensure that the following two conditions are invariantly true throughout program execution: SG is acyclic and a scheduled operation for a transaction does not violate the conditions of a legal schedule. This can be formally represented as follows:

$$<\text{SG is acyclic}> \tag{1}$$

$$<\neg\exists e, j(\text{Last\_op}_i(e, j) = \text{lock} \wedge \text{Last\_op}_i(e, j+1) = \text{lock})> \tag{2}$$

For an arc denoted by (T_{kl}, T_{st}) to be a member of E, it must be the case that all the operations of T_{kl} are completed before all the operations of T_{st}. This is represented as follows:

$$\begin{aligned}&<(T_{kl}, T_{st}) \in E_i \Leftrightarrow \exists\ e, j, i\ (\text{Last\_op}_i(e, j) = \text{lock} \wedge \text{Last\_op}_i(e, j+1) = \text{unlock} \wedge \\ &\text{Last\_tran}_i(e, j) = T_{st} \wedge \text{Last\_tran}(e, j+1) = T_{kl})>\end{aligned} \tag{3}$$

It is also necessary to ensure that if an entity is unlocked then the last scheduled operation for it is an unlock operation. This is represented as follows:

$$<lu_e \text{ is unlocked} \rightarrow \text{Last\_op}_i(e, 0) = \text{unlock}> \tag{4}$$

These four assertions form the invariant, I. The proof that the truth of I is preserved is found in [LuSM92]. Executable assertions are derived from the four assertions used to form the invariant I. These executable assertions are embedded into the transaction managers.

The details of the verification proof for a transaction manager process on processor i are not included in this paper. It turns out that it is similar to that of the lock manager and that the executable assertions derived are similar to that of the lock mananger.

B.2. Verification Details of Bitonic Sort

This section derives a error detecting bitonic sort algorithm. The transformation from a bitonic sort algorithm to a error detecting bitonic sort algorithm is based on the technique of transforming a verification proof outline of an algorithm to a error detecting algorithm. However, now, run-time efficiency is considered. This consideration of run-time efficiency has resulted in a error detecting algorithm that does not use all the assertions.

The target multicomputer interconnection topology considered in this paper is the popular hypercube topology. As mentioned above, these systems can grow to over 1000 processors. In general, the topology of an n-dimensional hypercube is a graph $G(P, E)$ with $N = 2^n$ vertices called nodes labeled $P_0, P_1, P_2, \ldots, P_{N-1}$. An edge $e_{i,j} \in E$ connects P_i and P_j if the binary representations of i and j differ in exactly 1 bit. If we let this bit position be k, then $P_i = P_{j \oplus 2^k}$. Thus, in an n-dimensional hypercube, each processor connects to n neighboring processors. Connections between the host and nodes are mainly used for program/data downloading and result uploading and are not represented in G. The algorithm used in this paper as a parallel sorting algorithm was introduced by Batcher in 1968[Batc68]. This bitonic sort algorithm was introduced as a parallel sorting algorithm that can take advantage of interconnection topologies such as the perfect shuffle and hypercube. There exists a bitonic sort algorithm that maps directly to a hypercube topology.

The postassertion {Q} of any sorting is defined in the following:

Definition B.2.1: *Given an input list* **I** = (I_i), *i=0,...,N-1 a sorting procedure S finds a permutation* $\prod = (\pi)$ *such that:*

$$I_{\pi_i} \le I_{\pi_{i+1}}, i = 0, \ldots, N-2$$

or

$$I_{\pi_i} \ge I_{\pi_{i+1}}, i = 0, \ldots, N-2$$

The general idea of a bitonic sort is to build up longer bitonic sequences which eventually lead to a sorted sequence.

Definition B.2.2: *A* **bitonic sequence** *is a sequence of elements* $O_0, O_1, \cdots, O_{N-1}$ *such that*

1. *There exists a subscript i,* $0 \le i \le N-1$ *such that* $O_0 \le O_1 \le \cdots \le O_{i-1}$ *and* $O_i \ge O_{i+1} \ge \cdots \ge O_{N-1}$

or

2. *There exists a subscript i,* $0 \le i \le N-1$ *such that* $O_0 \ge O_1 \ge \cdots \ge O_{i-1}$ *and* $O_i \le O_{i+1} \le \cdots \le O_{N-1}$

The fundamental operation in a bitonic sort is the compare-exchange operation, either min(x,y) or max(x,y).

Lemma B.2.1: [Batc68] *Given a bitonic sequence* $I_0 \le I_1 \le \cdots \le I_{N/2-1}$ *and* $I_{N/2} \ge I_{N/2+1} \ge \cdots \ge I_{N-1}$, *each of the subsequences formed by the compare-exchange steps:*

$$\min(I_0, I_{N/2}), \min(I_1, I_{N/2+1}), \cdots, \min(I_{N/2-1}, I_{N-1})$$

and

$$\max(I_0, I_{N/2}), \max(I_1, I_{N/2+1}), \cdots, \max(I_{N/2-1}, I_{N-1})$$

is bitonic with the property that $O_i \le O_j$ *for all* $i = 0, 1, \ldots, N/2-1$ and $j = N/2, N/2+1, \ldots, N-1$

For this presentation, for simplicity, we assume that $N = 2^k$ for some k and therefore the initial midpoint is N/2. Since each compare-exchange involves only a comparison between elements whose subscripts differ on only one bit and the number of elements is always 2_k, if we have one element per processor, then the bitonic sort can be easily implemented on a hypercube of dimension $n = \log_2 N$ [Quin87]. As a notational convenience, we define the following:

Definition B.2.3 *The home subcube* $SC_{i,j}$ *of dimension* i *of a processor* P_j *is the subcube of size* 2^i *that begins with processor* P_k, $k = j - (j \textbf{ mod } 2^i)$ *and includes all processors through* P_l, $l = j - (j \textbf{ mod } 2^i) + 2^i - 1$. *Let* $SC^S_{i,j}$ *denote the index* k *and* $SC^E_{i,j}$ *denote the index* l.

The following definition of λ reflects the communication patterns that naturally exist in the bitonic sort.

Definition B.2.4: *Define* λ^l_{ij}, *where* $i > 0$, *as follows:*

If l **mod** $2^{j+1} < 2^j$ **then**

$$\lambda^l_{ij} = \{l, l + 2^j\} \qquad j = i$$
$$\lambda^l_{ij} = \lambda^l_{i,j+1} \cup \lambda^{l+2^j}_{i,j+1} \qquad 0 \le j < i$$

and **if** l **mod** $2^{j+1} \ge 2^j$ **then**

$$l\lambda^l_{ij} = \{l, l - 2^j\} \qquad j = i$$
$$\lambda^l_{ij} = \lambda^l_{i,j+1} \cup \lambda^{l-2^j}_{i,j+1} \qquad 0 \le j < i$$

The Bitonic sort algorithm, instrumented with the assertions necessary for verification, is shown in Figure B.1 for each node node. Of particular interest are assertions Loop_i which asserts that at each execution of the outer iteration, a bitonic sequence in the subcube of dimension $SC_{i,node}$ is made.

```
**Procedure** Bitonic Sort;
  $ia_{node} = a$;
  $a_{node} = a$;
  $pa_{node} = a$;
  **for** i:=0 **to** n-1 **do**
  $< Loop_i >$ (assignment, consequence)
  $pa_{node} = a$;
  $a'_{node} = a$;
    {**for** j:=i **downto** 0 **do**
    $< Loop_j \wedge Loop'_j \wedge a'_{node} = a_{node} >$ (assignment, consequence)
      {$d:=2^j$;
      $< Loop'_j \wedge a'_{node} = a_{node} \wedge d = 2^j >$ (assignment)
       **if** (node **mod** (2d)<d)
      $< Loop'_j \wedge a'_{node} = a_{node} \wedge d = 2^j >$ (consequence)
        {**read** into data from node + d;
          $< Loop'_j \wedge a'_{node} = a_{node} \wedge d = 2^j \wedge data = a_{node+d} >$ (communication)
        **if** (node **mod** $2^{i+2} < 2^{i+1}$)
        $< Loop'_j \wedge a'_{node} = a_{node} \wedge d = 2^j \wedge data = a'_{node+d} \wedge$ node mod $2^{i+2} < 2^{i+1} >$ (alternative)
          {b := max(data,a);a := min(data,a);}
        $< Loop'_j \wedge d = 2^j \wedge data = a'_{node+d} \wedge$ node mod $2^{i+2} < 2^{i+1} \wedge b = \max(a'_{node+d}, a'_{node}) \wedge$
                $a = \min(a'_{node+d}, a'_{node}) \wedge \forall l_{SC^S_{j,node} \leq l \leq SC^E_{j,node}} [a \leq \max(a'_l, a'_{l+d})] >$ (assignment, consequence)
        **else**
      $< Loop'_j \wedge a'_{node} = a_{node} \wedge d = 2^j \wedge data = a'_{node+d} \wedge \neg($node mod $2^{i+2} < 2^{i+1}) >$ (alternative)
          {b := min(data,a);a := max(data,a);}
        $< Loop'_j \wedge d = 2^j \wedge data = a'_{node+d} \wedge \neg($node mod $2^{i+2} < 2^{i+1}) \wedge b = \min(a'_{node+d}, a'_{node}) \wedge$
        $a = \max(a'_{node+d}, a'_{node}) \wedge \forall l_{SC^S_{j,node} \leq l \leq SC^E_{j,node}} [a \leq \min(a'_l, a'_{l+d})] >$ (assignment, consequence)
        **write** from b to node+d;}
      **else** /* Send to neighbor - we are inactive this iteration */
                  /* Proof outline is symmetrical */
        {**write** from a to node-d;**read** into a from node-d;}
          $a_{node} = a$;
          $a'_{node} = a_{node}$;
      }   /* End for j */
    }   /* End for i */
```

Figure B.1: Bitonic Sort Instrumented with Verification Assertions

$Loop_i$ states that the values of the local variables a after each loop execution produces a bitonic subsequence in each subcube of size 2^{i+1}. $Loop_i$ also states that the values of the local variables a after each loop execution are a permutation of the previous values of the local variables a. Formally, $Loop_i$ is the following:

$$i \neq 0 \rightarrow (\text{node mod } 2^{i+1} < 2^i \rightarrow a_{SC^S_{i,node}} \leq \cdots \leq a_{SC^E_{i,node}} \wedge \text{node mod } 2^{i+1} \geq 2^i \rightarrow a_{SC^S_{i,node}} \geq \cdots \geq a_{SC^E_{i,node}}) \wedge$$
$$\exists l_{0 \leq l \leq N-1} a_{node} = ia_l \wedge \exists l_{SC^S_{i,node} \leq l \leq SC^E_{i,node}} a_{node} = pa_l$$

and Loop_j is the following:

$$\forall m_{j \le m \le i-1}\ [\text{node mod } 2^{i+2} < 2^{i+1} \rightarrow \forall k_{SC^S_{m+1,node} \le k \le SC^E_{m+1,node}}\ \forall l_{SC^S_{m+1,node+2^{m+1}} \le l \le SC^E_{m+1,node+2^{m+1}}}\ [a_k \le a_l] \wedge$$

$$\text{node mod } 2^{i+2} \ge 2^{i+1} \rightarrow \forall k_{SC^S_{m+1,node} \le k \le SC^E_{m+1,node}}\ \forall l_{SC^S_{m+1,node-2^{m+1}} \le l \le SC^E_{m+1,node-2^{m+1}}}\ [a_k \ge a_l])] \qquad \wedge$$

$$i = j \rightarrow (\text{node mod } 2^{i+2} < 2^{i+1} \rightarrow a^S_{SC_{i,node}} \le \cdots \le a^E_{SC_{i,node}} \wedge \text{node mod } 2^{i+2} \ge 2^{i+1} \rightarrow a^S_{SC_{i,node}} \ge \cdots \ge a^E_{SC_{i,node}}) \qquad \wedge$$

$$\text{node mod } 2^{i+2} < 2^{i+1} \rightarrow (\forall l_{SC^S_{i,node} \le l < SC^E_{i,node}} [a_l \le a_{l+1}] \vee \exists l_{SC^S_{i,node} \le l < SC^E_{i,node}} \forall k_{SC^S_{i,node} \le k < l} [a_k \ge a_{k+1}] \wedge \forall k_{l \le k < SC^E_{i,node}} [a_k \ge a_{k+1}]) \qquad \wedge$$

$$\text{node mod } 2^{i+2} \ge 2^{i+1} \rightarrow (\forall l_{SC^S_{i,node} \le l < SC^E_{i,node}} [a_l \ge a_{l+1}] \vee \exists l_{SC^S_{i,node} \le l < SC^E_{i,node}} \forall k_{SC^S_{i,node} \le k < l} [a_k \le a_{k+1}] \wedge \forall k_{l \le k < SC^E_{i,node}} [a_k \le a_{k+1}]) \qquad \wedge$$

$$\exists l_{0 \le l \le N} a_{node} = ia_l \wedge \exists l_{SC^S_{i,node} \le l \le SC^E_{i,node}} a_{node} = pa_l$$

and the assertion Loop_j' is constructed by replacing in the assertion Loop_j each a_i with a_i'.

a_{node}, a_{node}' and ia_{node} where $0 \le \text{node} \le N-1$ are the auxiliary variables used in the proof outline. The postcondition is that the output list is a permutation of the input list and that it is sorted. To decrease complexity the precondition is assumed to include that $n > 1$. In order to make use of this simplification, it is assumed that no faulty behavior occurs in the execution of the first iteration of the outer loop. This helps in eliminating tedious but trivial cases.

It is assumed for the sake of simplicity that we have loop synchronization. This simplifies proofs of non-interference. As part of inner loop synchronization, assign a_{node} to a_{node}'. This is done before any check for loop invariants. Loop invariants are checked at beginning of loops, but loop indices are changed at end. This implies that the loop indices are changed at the end of a loop iteration before invariants are checked.

B.3. Branch and Bound Verification

The objective of the optimization is to find a minimal length path from the initial configuration to a state that is a solution. For the N-puzzle problem, the following optimization function is used:

Definition B.3.1: *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(i.e. sum of distances that each tile must be moved for the tiles to be in the appropriate places) of the configuration denoted by* s_i.

When f is a monotone nondecreasing function [KoSt74, SuMc91], 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 an optimal node. 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 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.

Before proceeding with the highlights of the verfication proof and the translation of a branch and bound algorithm, we need to make a minor extension to the GAA (HAA) proof system. The GAA and HAA proof systems 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 σ_{ij}, ρ_{ij}, where σ_{ij} is the set of all messages sent from process i to process j and ρ_{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 σ_{ij} and ρ_{ij} are immediately updated. It is also assumed that $\rho_{ij} \subseteq \sigma_{ij}$ is invariantly true throughout program execution.

The following auxiliary variables are used:

S_I: This is the set of all nodes in the tree that represent 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.

$solution_i$: This is the value of $s_{current}$ at the termination of worker i.

Each node in the tree is referred to as a state, s_i, of the system configuration where i is a unique integer. We will assume that s_0 denotes the state corresponding to the root node or 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. 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. If $s_i \in A_I$ then s_i is a solution or final configuration.

We need to assert a precondition, <P>, on the program which asserts that there is a solution reachable from the initial configuration s_0. In other words, P is as follows:

$$< A_I \neq \emptyset >$$

The parallel algorithm involves dividing the work in terms of subtrees and has many processors working on the problem simultaneously. A task refers to transforming some s_i to s_j by a legal move of type, m_k, where $m_k \in M$. 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. Formally, this corresponds to partitioning the auxiliary variable S_I into the disjoint sets $S_0, S_1, \cdots, S_{N-1}$, which also satisfies the following immediately after initial board distribution:

$$< S_0 \cup S_1 \cup \cdots \cup S_{N-1} = S_I > \tag{1}$$

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. Since task migration does not change the set of nodes to be examined as a whole, migration only changes the set of nodes to be examined by the migrating and receiving worker processes. 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 following set in the assertion (1)

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

From the satisfaction proof that must be done at each task migration, we know that $< s_{task} \in S_I >$and $\sigma_{ji} \subseteq S_I$ is true in any migrating process. Since $\rho_{ji} \subseteq \sigma_{ji}$ is invariantly true, we see that $< S_i \subseteq S_I >$.

We have asserted that each worker process i examines those nodes that are part of the state space when it receives a task. We also need to assert that we don't lose portions of the search space by pruning and that the algorithm only examines those nodes that are in the state space. This leads to the following invariant:

$$< S' \cup S = S_I \wedge A' \cup A = A_I > \tag{2}$$

The updating is done by determining all the set of nodes associated with the pruned subtree by finding all the reachable nodes

$$T_0 = \{ s_j \mid s_j \text{ is a reachable configuration of } s_i \}.$$

from the root node s_i of the subtree to be pruned off and determining all the solution nodes in the pruned subtree. The solution nodes are deleted from A and added to A'. The state space is updated by $S_i = S_i - T_0$. It is, therefore, easy to see that $S_i \subseteq S_I$ is still true.

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_I, 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. The current solution, $s_{current}$ which has the lowest bound then contains the optimal moves for the puzzle.

The above discussion leads us to the precondition, Pre_i to each worker process i, which states that each worker processes initially have no tasks to examine and no communication has taken place. This is represented by

$$< S_i = \emptyset \wedge SB_{ij} = \emptyset \wedge RB_{ij} = \emptyset \quad \text{for all } j, 0 \le j \le N-1, j \ne i >$$

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

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

For each terminating component process labelled node, s_{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 Q is represented as follows:

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

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

$$< s_{current} = \min(a \mid a \in R_i, \text{ where } R_i = \bigcup RB_{ij}) > \quad (3)$$

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).

$$< SB_{ij} = R_{ji} \bigcup \{x \mid x \in \sigma_{ij} - \rho_{ij} \wedge x \text{ is a solution}\} > \quad (4)$$

$$< SB_{i0} = \cdots = SB_{iN-1} > \quad (5)$$

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:

$$< \text{For all } x \in A', \text{ there is an i and y such that } y \in R_i \text{ and } f(y) \leq f(x) > \quad (6)$$

The formal verification now goes onto formulate loop invariants and that termination shows the postcondition. Since the details of this are unimportant in the transformation process, we omit this discussion here.

We will now discuss the concept of a naturally redundant algorithm and then it will be seen why the redundancy 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 divided in *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 broadcasted solutions are the output of each phase, where each phase is defined as being between broadcasted 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 B.3.1: *The N Puzzle algorithm of this section annotated with the the executable assertions developed in this paper 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 $s_{current}$ (where $s_{current}$ 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 $s_{current}$ 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.

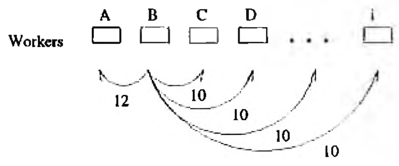

Figure B.2: Bound received by worker A is greater than all the others.

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.

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.

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 error-detecting program developed in this section is naturally redundant, hence, there is no need for explicit consistency.

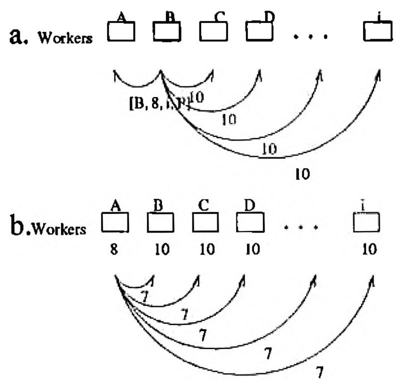

Figure B.3 a. The bound received by worker A is lower than all other workers. b. Worker A self corrects itself when a lower bound is discovered.

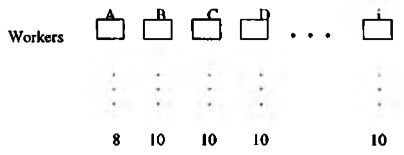

Figure B.4: Bound received by worker A is lower than all other workers at the completion of the first round.

BIBLIOGRAPHY

[AmKB89] Ammann, Paul E., Knight, John C., and Brilliant, S. S., "The Influence of Testing on Fault-Tolerant Software," *Proc. 12th Int'l Symposium on Fault-Tolerant Computing,* [June 26-28, 1990, pp. 408-415].

[ApRo81] Apt, R. and Roever, W., "A Proof System for Communicating Sequential Processes," *ACM Transactions on Programming Languages and Systems*, 2, 3, 1981, 359-385.

[BaKP85] Barringer, H., Kuiper, R., and Pnueli, A., "A Compositional Temporal Approach to a CSP-like Language," *Proceedings of the IFIP Conference, The Role of Abstract Models in Information Processing,* Elsevier Scientific Publishers, February, 1985, pp. 209-229.

[Batc68] Batcher, K., "Sorting Networks and Their Applications," *Proc. of the 1968 Spring Joint Computer Conference*, vol. 32, AFIPS Press, Reston, VA, pp. 307-314.

[BeGo82] Bernstein, P. and Goodman, N., "Concurrency Control in Distributed Database Systems," *Computing Surveys*, 13, 2, 1981, pp. 185-221.

[Clin73] Clint, M., "Program proving: coroutines," *Acta Informatical*, 2, 1973, pp. 50-63.

[Dijk76] Dijkstra, E., *A Discipline of Programming*, 1976, Prentice-Hall, Inc.

[EGLT76] Eswan, K.P., Gray, J.N., Lorie, R.A., and Traiger I.L., "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, 19, 11, 1976, pp. 624-633.

[Hoar69] Hoare, C., "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, 12, 10, 1969, 576-583.

[Hoar78] Hoare, C., "Communicating Sequential processes," *Communications of the ACM*, 21, 1978, 666-677.

[HuAb84] Huang, K. and Abraham, J., "Fault-Tolerant Algorithms and Their Application to Solving Laplace Equations," *Proceedings of the 1984 International Conference on Parallel Processing*, August 1984, pp. 117-122.

[HuAb86] Hua, K. and Abraham, J., "Design of Systems with Concurrent Error Detection using Software Redundancy," Proceeding Fall Joint Computer Conf. Nov. 2-6, 1986, pp 826-835.

[HoMc91] Hong, C.E. and McMillin, B.M., "Fault-Tolerant Parallel Matrix Multiplication with One Iteration Fault Detection Latency," *to appear in Compsac 1991*

[KoSt74] Kohler, W., and Steiglitz, K., "Characterization and Theoretical Comparison of Branch-and-Bound Algorithms for Permutation Problems," *Journal of the ACM*, vo. 21, no. 1, 1974, pp. 140-156.

[LaLi92] Jean-Claude Laprie and Bev Littlewood, "Probabilistic Assessment of Safety-Critical Software: Why and How?," *Communications of the ACM*, Vol. 35, No. 2, 1992, pp. 13-21.

[LaMG91] Larangeria, L., Malek, M., and Jenevein, J., "On Tolerating Faults in Naturally Redundant Algorithms," *Tenth Symposium on Reliable Distributed Systems*, 1991, pp. 118-127.

[LaSP82] Lamport, L., Shostack, R. and Pease, M., "The Byzantine General's Problem," *ACM Transaction on Programming Language Systems,* vol. 4, July 1982, pp. 38 2-401.

[LeGr81] Levin, G.M and Gries, D., "A Proof Technique for Communicating Sequential Process," *Acta Information*, 15, 1981, 281-302.

[LiMc92] Liu, J. and McMillin, B., "A Divide and Conquer Ring Embedding Scheme on Hypercubes with Efficient Recovery Capability," *Proceedings of the 21st International Conference on Parallel Processing,* (with J. Liu) (to appear) 1992.

[LuMc91] Lutfiyya, H. and McMillin, B., "Comparison of Three Axiomatic Proof Systems," *UMR Department of Computer Science Technical Report Number CSC 91-13*, (Submitted to *Parallel Processing Letters*)

[LuSM92a] Lutfiyya, H., Schollmeyer, M., and McMillin, B., "Fault-Tolerant Distributed Database Lock Managers," *UMR Department of Computer Science Technical Report Number CSC 92-05*, (Submitted to the 14th International Conference on Software Engineering)

[LuSM92b] Lutfiyya, H., Schollmeyer, M., and McMillin, B., "Fault-Tolerant Distributed Sort Generated from a Verification Proof Outline," *Second International Workshop on Responsive Computer Systems, 1992* (To Appear)

[LuSM92c] Lutfiyya, H., Sun, A., and McMillin, B., "A Fault-Tolerant Branch and Bound Algorithm Derived from Program Verification ," *IEEE Computers Software and Applications Conference(COMPSAC), 1992* (To Ap pear) Also as *UMR Department of Computer Science Technical Report Number CSC 92-02*

[MaML83] Mahmood, A., McCluskey, E. J., and Lu, D. J., "Concurrent Fault Detection using a Watchdog Processor and Assertions," IEEE 1983 Int'l Test Conf. pp 622-628.

[McNi92] McMillin, B. and Ni, L., "Reliable Distributed Sorting Through The Application-oriented Fault Tolerance Paradigm," *IEEE Trans. On Parallel and Distributed Computing,* 1992, (to appear).

[Mili81] Mili, A., "Self-Checking Programs: An Axiomatisation of Program Validation by Executable Assertions," 11th Annual Int'l Symp. on Fault Tolerant Computing, 1981, pp 118-120.

[Mili85] Mili, A., "Towards a Theory of Forward Recovery," *IEEE Transactions on Software Engineering,* [vol. SE-13, no. 1, , 1987, pp. 23-31.]

[OsKA86] Osawa, G., Kawai, T., and Aiso, H., "Fault Tolerant Scheme on Partial Differential Equations," *Proceedings of the 1986 International Conference on Parallel Processing, August 1986, pp. 413-416.*

[Owic75] Owicki, S., *Axiomatic Proof Techniques for Parallel Programs,* Computer Science Dept., Cornell University, PhD Thesis, 1975.

[OwGr76] Owicki, S. and Gries, D., "An Axiomatic Proof Technique for Parallel Programs I," *Acta Informatica,* 6, 1976, 319-340

[OwLa82] Owicki, S. and Lamport, L., "Proving Liveness Properties of Concurrent Programs," *ACM TOPLAS,* Vol. 4, No. 3, July 1982, pp. 455-495.

[PeSL80] Pease, M., Shostak, R. and Lamport, L., "Reaching Agreement in the presence of faults," *Journal of the ACM,* Vol 27, No. 2,, April 1980, pp. 228-234.

[Pnue84] Pnueli, A., "In Transition from Global to Modular - Temporal Reasoning about Programs," *Logics and Models of Concurrent Systems,* edited by K. Apt, Nato ISI Series. Vol. F13, pp. 123-144.

[Quin87] Quinn, M., *Designing Efficient Algorithms for Parallel Computers,* McGraw-Hill, New York, 1987.

[Rand75] Randall, B., "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering,* Vol. SE-1, No. 2, June 1975, pp. 220-232.

[Renn84] Rennels, D., "Fault-Tolerant Computing-Concepts and Examples," *IEEE Transactions on Computers,* Vol C-33, No. 12, , Dec. 1984, pp. 1116-1129.

[Reit87] Reiter, R., "A Theory of Diagnosis From First Principles," *Artificial Intelligence,* Vol. 32,, 1987, pp. 57-95.

[Roma87] Roman, G., "Specifying Software/Hardware Interactions in Distributed Systems," *Proceedings of the 9th International Conference on Software Engineering,,* March 1987, pp. 126-139.

[ScSc84] Schlichting, R., and Schneider, F., "Using Message Passing For Distributed Programming: Proof Rules and Disciplines,"

[SuMc91] Sun, A. and McMillin, B., " Application-Oriented Fault-Tolerant Parallel Branch&Bound," UMR Department of Computer Science Technical Report CSC-91-014, September, 1991.

[Soun84] Soundararahan, N., "Axiomatic Semantics of Communicating Sequential Processes," *ACM Transactions on Programming Languages and Systems,* 6, 4, 1984, 647-662.

[Vorn87] Vorberger, O., "Fault Tolerant Parallelization of Branch&Bound Algorithm," *The 7th Distributed Computing Symposium,* 1987, pp. 194-197.

[YaCh75] Yau, S. S. and Cheung R. C., "Design of Self-Checking Software," Proc. Int'l Conf. on Reliability Software, April 1975, pp 450-457.

[YaHa84] Yanney, R. and Hayes, J., "Distributed Recovery in Fault Tolerance Multiprocessor Networks," *4th International Conference on Distributed Computing Systems*, IEEE 1984, pp. 514-525.