AN EFFICIENT FAULT TOLERANT DISTRIBUTED SHORTEST PATH PROGRAM IN ADA

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston - University Park

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Camron Malik

December, 1987

ACKNOWLEDGEMENT

I dedicate this thesis to my family, without their love and support none of this would have been possible. To my father and mother, no one could ask for better parents. To my wife who has been my rock of Gibraltar. Finally, to my most precious Ike, for whom this has all been.

I would also like to take this opportunity to thank my committee members Dr. Stephen Huang and Dr. Tiee-Jian Wu. To Dr. Pen-Nan Lee for his help, understanding and friendship I am especially grateful. His guidance was appreciated and his trust in me was a source of pride.

AN EFFICIENT FAULT TOLERANT DISTRIBUTED SHORTEST PATH PROGRAM IN ADA

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston - University Park

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Camron Malik

December, 1987

ABSTRACT

A simple, elegant algorithm upon implementation presents innumerable problems. This paper provides insight into the difficulties of implementing a distributed algorithm. This is followed by an efficient, fault tolerant implementation of the Distributed Shortest Path Algorithm. The provision of fault tolerance has a large overhead in terms of the number of messages required. A modification of the algorithm is proposed to reduce the number of messages, using buffering in conjunction with Ada constructs to achieve this in the implementation.

The unrestricted communication in a distributed system produces situations conducive to deadlock. This is particularly true if a synchronous form of message passing is used, as processes will wait indefinitely for each other. To ensure freedom from deadlock a variant of nondeterministic message sending based on Ada timed out entry calls is used. Distributed programs are also, by virtue of their complexity, difficult to verify. Even after extensive testing residual design inadequacies may be present. Thus the concept of Communication Closed Layers is used to design the program. The Consensus-Global Tester is used to implement error detection and assist in error recovery. In the event of an error, a Backward error recovery scheme is used which saves the essential information. Thus, computation can be reinitiated using the saved values.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

The trend towards distributed processing on computer networks has led to an increase in the number of distributed algorithms and the development of programming languages to exploit the concurrency. But two major issues have not yet been addressed. The first issue concerns the problems associated with the implementation of the algorithms, within the constraints of the languages. The second issue concerns the assurance of reliability in such a complex software system, as the results depend on the unpredictable order in which actions from different processes are executed. In this paper we consider the problems and drawbacks of implementing the Distributed Shortest Path Algorithm [CHAN82] within the constraints placed by a language, specifically Ada*. We then design and implement a fully distributed, fault tolerant program which meets all correctness criteria.

The Distributed Shortest Path Algorithm is an elegant distributed solution to compute the shortest path from a special vertex v_1 to all other vertices of a weighted, directed graph. The unrestricted communication in a distributed program and the unpredictable order of execution of the component processes pose problems. These are compounded by the constraints placed by a language. Thus to achieve freedom from deadlock requires either an indirection methodology or the use of

*Ada is a registered trademark of the U.S. Govt.,Ada Joint Program Office (AJPO)

variant Ada constructs to provide nondeterminism on output. A general method for overcoming deadlock is proposed and implemented using Communication Processes (buffers).

Distributed programs are inherently difficult to verify and even after extensive testing, may have residual design errors. Thus techniques for designing correct programs have to be utilized. This particular fault tolerant implementation is based on the concept of Communication Closed Layers [ELRA83], which partitions programs logically / physically to provide what are called Safe Layers. Such a design methodology coupled with the concept of Consensus-Global Testers [LEE88] provides fault tolerance. Hence, error detection and recovery are possible.

A Recovery Block [RAND75] type scheme is used to implement error detection and recovery. The premise of a Recovery Block type scheme is that errors will occur, thus "spare" modules must be provided. Hence, at the conclusion of a particular computation, if an error is detected the "spare" can be used to recompute the values. While the erroneous values are discarded. The errors are detected through the use of a Tester module which assures that the results are either "acceptable" or erroneous.

The ultimate objective is to maximize concurrency and provide fault tolerance without incurring overheads in time-space. To begin the discussion a brief outline of the Distributed Shortest Path Algorithm and the complete backgrounds on the concepts, techniques and methodologies will be given in section 2. This will be followed in section 3 by a description of the implementation and its algorithm.

Section 4 is devoted to the analysis. Finally, in section 5, some concluding remarks are made.

II. BACKGROUND

In this section we provide the conceptual background of the techniques which form the basis for this thesis. These are firstly, the language of implementation, Ada. Secondly, the concept of fault tolerance followed by the Safe Layering design methodology. Then a consideration of the problems and difficulties of implementing a distributed program are provided. Finally, an overview of the distributed algorithm is given. But before a complete exposition of the theory is given, a few basic definitions are in order.

To begin with, a concurrent program specifies two or more sequential programs that may be executed concurrently as parallel processes. These processes communicate and synchronize, in order to cooperate to achieve a common objective. But the absolute or relative speeds of execution of the component processes are unpredictable.

Concurrent programs may be executed in several different environments, depending basically on the availability of processors and their interconnections. The first method allows processes to share one or more processors and is referred to as, multiprogramming. If each process is executed on a single processor, but all processors share a common memory, it is referred to as multiprocessing. Finally, the execution of processes on dedicated processors connected by a network is called distributed processing. Since no memory is shared cooperation is achieved through message passing or remote procedure calls. Thus a distributed program consists of a

collection of processes or tasks executed in a distributed processing environment.

In what follows, the terms task and process are interchangable and refer to self sufficient execution units which communicate via messages.

2.1 Concurrent Ada

Ada, the new general purpose programming language, is based on definitions proposed by the US Department of Defense for use in embedded Systems. It is the culmination of a decade of specification and revision of successive versions of the language and reflects the current trend towards data abstraction, multitasking, generics, exceptions handling, readability, reliability, etc.

In many circumstances programs have to be written as several parallel activities which communicate / synchronize in order to cooperate. In Ada this parallelism is described by means of tasks, which is a unit of concurrency. When two tasks need to interact they do so through a mechanism known as a rendezvous. A rendezvous takes place when one task calls an entry declared in another. Each entry has a corresponding ACCEPT statement. For example,

.

.

.

ACCEPT message(messagetype : IN INTEGER) DO

.

.

.

END;

.

.

.

P1.message(4);

accepts a message and subsequently sends a message to task P1, which has a corresponding ACCEPT.

The body of the ACCEPT statement acts as a critical section and no other communication can take place nor can any values be changed in the calling task until the conclusion of this rendezvous. The end of the rendezvous coincides with the END statement of the ACCEPT, in the called task. If the ACCEPT has no body or parameters it acts as a synchronization primitive only and no information is exchanged. The use of parameters allows information to be exchanged by reference or value.

A strength of Ada which is not as apparent, in contrast to its better known features, is its flexibility. Though the specification of Ada do not explicitly provide for nondeterministic output, the use of timed out entry calls allow the mimicking of such nondeterminism. The timed out entry call allows a sequence of statements to be executed alternatively, if an entry call is not accepted within the specified duration. Thus

```
SELECT
   P1.message("entry call");
OR
   DELAY X;
   -- Statements
END SELECT;
```

will execute the statements following DELAY X, if task P1 does not accept the call within X seconds. This ensures that the calling task will not wait indefinitely, if the destination task cannot accept the call.

When a task calls another, Ada relies on an asymmetric naming scheme to implement general entry points. That is, the calling task needs to know the identity of the called task. But the called task (server) is not required to know who the caller (user) is. Thus, entry points can be called by any process without the server requiring prior knowledge of the users identity. If a task needs to know whether processes are waiting to rendezvous it utilizes the concept of Attributes. The syntax is P'COUNT, which provides the number of tasks waiting at entry point P.

2.2 Software Fault Tolerance

The need to provide increased reliability in computer system led to the approach of achieving this goal through the use of fault prevention. Reliance is placed on tools and techniques such as verification, documentation, testing, etc.

Such techniques assume that all possible causes of unreliability can be removed prior to delivery and reliance will not be placed on a system until all "bugs" have been removed. This approach fails to account for faults which were unanticipated and thus not weeded out during the design and testing of the system. It is reasonable to assume faults may be present in a system and will have to be tolerated. Thus the concept of fault tolerance uses redundancy of design as a means to provide error detection and recovery from residual design inadequacies. This ensures uninterrupted service even in the event of faults. To achieve this objective, fault tolerant systems must detect errors, assess the damage, try to recover and provide continuous service.

Two complementary approaches for providing fault tolerance in software have evolved. These are forward error recovery and backward error recovery. The aim of forward error recovery is to identify the error and based on the available knowledge correct the system state to provide continued service. An example of such an approach is N-Version Programming. In contrast, backward error recovery manipulates the system state so as to achieve a "reversal of time". That is, to a state prior to the erroneous one without regard for the current state. Thus previous states are saved on a stable medium, to be recalled if the need ever arises.

The recovery block scheme [RAND75] is an example of a backward error recovery technique and like all fault tolerant schemes relies on redundancy. It consists of three distinct parts: a recovery point, execution modules and an acceptance test point. The first of these is a point in the execution of a program when

the important variables are saved. This occurs prior to entering a recovery block. The second part consists of a primary module, which is executed first upon entering a recovery block. Upon completion the process must pass an acceptance test to ensure the reliability of its results. If the test is passed, then the process proceeds. But if the test is failed the process state is restored to its original version (saved on entering the recovery block). Then an alternate module of the program is executed, in the hopes that the alternate will not have the residual design inadequacies present in the primary.

The alternate blocks / modules may be of differing design, algorithms, languages or a combination thereof. The premise is that residual design inadequacies present in one module will not be present in another. Any number of alternates may be used as long as they provide a measure of fault tolerance within acceptable costs. For example, if four algorithms to solve a particular problem are available and their time complexities are n log n, n^2, n^3 and n^{12}, then the last version even though it provides redundancy, may be too expensive to employ, especially in a time constrained application.

The acceptance test is a last moment check to ensure the reasonableness of the output and is by no means a test for absolute correctness. This acceptance test is over and above the usual interface checks provided by the system - which lead to exceptions, etc. Thus if no exception has been raised and the output of the module meets the acceptance criteria it is assumed that no fault occurred.

2.3 Safe Layering

Distributed programs, by virtue of their complexity, are very difficult to verify formally. Even after extensive testing and debugging residual design inadequacies may be present. This coupled with the unrestricted communication between concurrent processes could cause the propagation of erroneous values. Ultimately leading to erroneous results or a crash of the software system. Thus there is a need for methodologies to design reliable programs and for techniques to detect and recover from faults. One such design method, based on the concept of Communication-Closed Layers proposed in [ELRA83], provides a means to design reliable distributed programs in what are termed Safe Layers. This in conjunction with the Consensus-Global Tester [LEE88] provides error detection and recoverability. The provision of fault tolerance based on these techniques does not give up any degree of concurrency, allowing component processes to execute at their own pace.

2.3.1 Safe Layers

Distributed programs can be viewed as having a two dimensional data-flow. That is, sequential within the process and parallel between processes. Thus, in order to design a distributed program we must consider the sequential behaviour within each process and manage synchronization / communication

between the processes. The concept of Safe Layering allows such a consideration. The basic idea is to view distributed programs as a sequential composition of concurrent Layers. For example, a concurrent program P consisting of interacting processes p_1 ; p_2 ; ·····; p_n is defined in CSP [HOAR78] syntax as :

$$P :: [\, p_1 \parallel p_2 \parallel \cdots\cdots \parallel p_n \,]$$

Furthermore, each component process can be subdivided into d logical / physical segments. Thus each process (p_i) may be defined as :

$$p_i :: [\, p_i^1; \cdots\cdots ; p_i^d]$$

Thus, in general, process segments can be defined as :

$$P_i^{seg} \quad \{ i = 1..n \quad , \quad seg = 1..d\}$$

and a $Layer^k$ is :

$$[\, p_1^k \parallel p_2^k \parallel \cdots\cdots \parallel p_n^k \,]$$

The Sequential Composition (denoted by ";") of a concurrent program P is $[\, Layer^1 ; \cdots\cdots ; Layer^d \,]$

This allows a concurrent program to be viewed as a collection of sequential layers. But gives up some concurrency and requires a global synchronization scheme, as commands in a following layer are not available until the previous layer has terminated.

The Distributed Composition (denoted by ":") of a concurrent program P is:

$$[\ Layer^1 : \cdots\cdots : Layer^d\ ]$$

and is exactly equivalent to :

$$[\ p_1{}^1; \cdots\cdots ; p_1{}^d \parallel \cdots\cdots \parallel p_n{}^1; \cdots\cdots ; p_n{}^d\ ]$$

Thus allowing a process to execute at its own pace without any global synchronization and ignoring layer boundaries.

The equivalence of the two compositions can be provided by assuming for all layers, that $Layer^k$ is Communication Closed. That is, in any communication both members must belong to the same layer. Thus if inter-layer communication is disallowed, across layer boundaries, each of the layers is communication closed and such layers are called Safe Layers. These Safe Layers can be used as units of modularity with layer boundaries serving as synchronization points [LEE88], [ELRA83], [GERT86], [MOIT83].

The Distributed Shortest Path Algorithm (DSPA) is implemented in two layers corresponding to the two phases of the algorithm, described in section 2.5.

2.3.2 Consensus-Global Tester

The efficacy of fault tolerance depends to a large extent on the ability to detect errors and consequently have a chance to correct the errors. Thus error detection is an extremely important phase in computation and relies heavily on the ability of the tester to "catch" the errors. In sequential programs the errors are isolated within single programs which are not affected by outside influences. But in

a distributed system, where many processes may be running concurrently and interacting, errors outside the module can affect the outcome. Some errors may be localized but, through interactions, have tainted parts of the program which appear to be fine.

A tester for a sequential program is required to ensure that specifications for a particular program are met. In the case of distributed programs, the tester must ensure the correctness of the results for the entire computation. This is made more difficult since the order of execution, of the actions of the interacting processes, are unpredictable. Thus, so are the results.

The Consensus-Global Tester [LEE88] based on the premise that there are interactions amongst processes provides error detection for all the component processes. This is achieved by providing a global specification, which can test the correctness of the results of all the interacting processes. In the event of a global error all tasks are required to rollback.

If a distributed implementation can be partitioned into regions or layers in such a way that error detection and recovery can be localized. Then the concept of Global Testers can be applied to each of the regions to regionalize the error detection and recovery, without having an adverse effect on the other regions. Thus, errors can be detected and recovery initiated only in those particular regions. In the event of an error, rollback and recovery occur within the region. But if no regional errors are detected the results are sent to the Global Tester for consensus-global testing. That is, to ensure that all regions meet the specification as a whole.

In the program to be implemented the concept of a single Consensus-global Tester for each phase of the computation is used. This tester should verify that a global assertion holds in all cases.

2.4 Problems and Difficulties

The concept of distributing processing is a powerful and useful one, but must be utilized with extreme care. Several problems are faced in the effort to implement a distributed program, and these issues have to be resolved to profit from the enormous potential of distributed processing. These issues include the danger of deadlock, unnecessary blockage, overheads of messages and processes, language constraints, reliability, debugging of errors and a fully distributed implementation. When addressing these issues compromises have to be made, which ultimately affect the implementation and its efficiency.

2.4.1 Deadlock Problem

In distributed solutions, the unrestricted inter-process communication produces situations conducive to deadlock. For example, some arbitrary process P_i attempts to communicate with another process P_j; simultaneously P_j may try to send a message to P_i. This circular wait situation is unresolvable as both processes would

wait indefinitely for the other to receive its message. There are two basic solutions to this problem, either deadlock avoidance or deadlock detection and arbitration. The latter is much more costlier in terms of the overhead of monitoring and is almost impossible to achieve, in general, for distributed programs. The avoidance of deadlock is relatively easier to achieve through careful structuring and design of the program [LEE87]. But requires some degree of intuition on the part of the programmer and flexibility in the programming language.

A general method to ensure freedom from deadlock makes use of indirection during communications. This is achieved through the use of buffer processes, which buffer and redirect messages, thus circumventing the need for direct communication. A second technique depends on the flexibility of the language Ada to provide nondeterminism on output.

2.4.2 Blocking Problem

A less serious but equally important issue concerns unnecessary blockage / waiting. A process blocked for communication / synchronization must not have to wait too long. This issue gains significance if it is realized that the speeds of execution of processes are arbitrary and therefore unpredictable. Thus a faster executing process may have to wait for a slower partner to effect a synchronization or complete a communication attempt. To alleviate this problem a process must, upon finding the called process busy, be allowed to continue processing on

something else. Thus valuable computing power is not lost waiting for events to occur. For example, if a process P_1 attempts to communicate with a process P_2 and finds P_2 busy. P_1 should not be required to wait for P_2, instead P_1 may delay a short time and thereafter proceed on its own, subsequently returning to reattempt a rendezvous.

2.4.3 Message and Process Overheads

Since processes are executed on systems which could be geographically separated and no sharing of memory occurs, the only means of communications are remote procedure calls or message passing. In the algorithm and the implementation language, message passing is assumed and thus only the latter is considered. It is apparent that communication through messages has a substantial overhead in terms of the delay, the amount of memory required to buffer message and the number of messages propagated. Thus any implementation should include the reduction of messages as an a priori requirement. If such a reduction is possible through the implementation, by modification to the original algorithm or use of concepts and constraints, then the necessary steps must be taken to ensure minimality of the number of messages.

Aside from the number of messages, under certain circumstances, the number of processes may be quiet high. These processes may be needed for

secondary purposes, such as buffering. They should be kept to a minimum, or eliminated altogether if possible. Since the overhead lies not only in the number of processes but also for inter-process communication. The inefficiency inherent in a system using a large number of processes and / or messages is a drain on the system. This ultimately affects performance and throughput of the system is reduced.

2.4.4 Language Constraints

Until the recent development of general purpose programming languages, which incorporate multitasking and constructs for concurrency as primitives, most languages did not provide for such concepts. But the provision of such capabilities in the new languages is by no means complete, as they are still not powerful or expressive enough to allow all types of implementations. In the event that a construct is not directly available to the programmer, the flexibility of a language plays an important role in allowing solutions without incurring unacceptable overheads. An example is the timed out entry call in Ada, without which nondeterminism for output messages would not be possible.The inventors of distributed algorithms usually do not consider specific languages to implement their algorithms. Therefore, these algorithms are not always amenable to implementation within the constraints of a language. The tools provided by a language, either directly or indirectly, may be utilized by a programmer in cases where regular

constructs are too confining or inadequate.

2.4.5 Reliability Problems

There are two types of correctness properties which all programs must possess - Safety and Liveness. Safety properties are the static portion of the specifications and are explicitly stated. An example is mutual exclusion. Liveness deals with the dynamic properties and ensures that an event will eventually happen. Deadlock is an example of a breach of liveness. These issues are extremely important in concurrent programs as the results of the execution of several processes depends on the order in which actions from different processes are executed. The complexity of the situation greatly increases the probability that the programmer will make mistakes and that errors will not be detected during testing. Such design errors would ultimately lead to the violation of the correctness properties and either incorrect results or, failure of the software system. Until reliable proofs of correctness which cover implementation details are available for realistic software, reliance has to be placed on design methodologies and software fault tolerance.

A secondary issue concerned with reliability is that of debugging. This is an example of fault prevention and can be useful in finding and removing some errors, which would cause unnecessary wastage of computing power. For example, during software testing the ease of readability of a program is essential and enhances the chances of catching and fixing errors. But most languages seem to consider

readability as an afterthought. If a language is too terse (such as APL) reading it is difficult and finding logical errors next to impossible. On the other hand a language which is too prolix affects the programmer just as badly.

2.4.6 Distributed Implementation

It is obvious that a distributed program must be exactly that, distributed. Since the quality, speed and efficiency all stem from the distributed environment which allows various parts of a concurrent program to execute at their own pace. It is possible to implement distributed algorithms using a host or controller process to restrict communication . But this reduces concurrency and has a detremental effect on the speed, efficiency and ultimately the quality of the program. A centralized model using a single controlling process is infeasible, not only for the reasons above, but it is prone to bottlenecks and intolerant to faults. The loss of the central node can cause a crash of the entire system. Such an implementation would also sequentialize a distributed algorithm, making it no better than a sequential program given time slices on a single processor. Thus all distributed programs must allow unrestricted communication without any host or controller process and use the advantages provided by the language, the algorithm and the system.

2.5 Distributed Shortest Path Algorithm

In this section we provide the background and highlights of the Distributed Shortest Path Algorithm. The complete algorithm can be found in appendix A.

The algorithm implemented is an elegant, distributed solution to compute the shortest path from a vertex to all other vertices of a weighted, directed graph in the presence of negative cycles. A directed graph G = (V,E) consists of 2 sets. V is a set of vertices and E is a set of edges. If an edge $<v_i,v_j>$ is incident to vertices v_i and v_j, then a path exists from v_i to v_j. The vertex v_i is called the predecessor of v_j and v_j is the successor of v_i. Each edge has associated with it a length l_{ij} corresponding to the distance from v_i to v_j. In the event a length l_{ij} is negative, a cycle of negative length may exist. Consequently, all vertices reachable from the negative cycle will have l_{ij} equal to $-\infty$. An example of such a graph is shown in figure 1.

In this algorithm processes communicate through messages and the presence of message buffers is assumed. The computation is done in two phases. The first phase computes the minimum distance from vertex v_1 to all other vertices. If there is a negative cycle a vertex will have a distance of $-\infty$. The second phase is used to inform the vertices that they are at a distance of $-\infty$. In phase I the path lengths are propagated using a length message and successors reply using an

acknowledgement message.Where there is no ambiguity the terms vertex and node will be used interchangably.

Process P_1 at Node v_1 initiates Phase I by using length messages to inform its successors of its distance from them. The successors upon receiving this value add the distances to their respective successors (to the received value) and pass on the new value.

This iterates until all successors receive their respective length messages. Upon the receipt of a length message each process updates its local value for the shortest path received thus far from a predecessor and propagates the message. An acknowledgement sent in response to a length message is used to terminate phase I.

Phase II, again initiated at node v_1, employs two types of messages. Namely the over- and over? messages. An over- message is sent if it is determined that a negative cycle exists, i.e. shortest path distance is $-\infty$. The receipt of an over- message requires a successor to set distance to $-\infty$, unless it already has distance equal to $-\infty$. The over- message is then propagated. The second message type, an over?, is sent if it has not been determined whether distance is $-\infty$. In the event that there are no outstanding acknowledgements the successor propagates the over?. But, if some length messages remain to be acknowledged, an over- is sent.

The algorithm assumes each process has a queue-like input buffer, to which messages from its neighbors are appended. Since Ada does not support such a capability, one implementation buffers outgoing messages at the source of the

communication. The other uses variants of Ada constructs to provide nondeterminism on output.

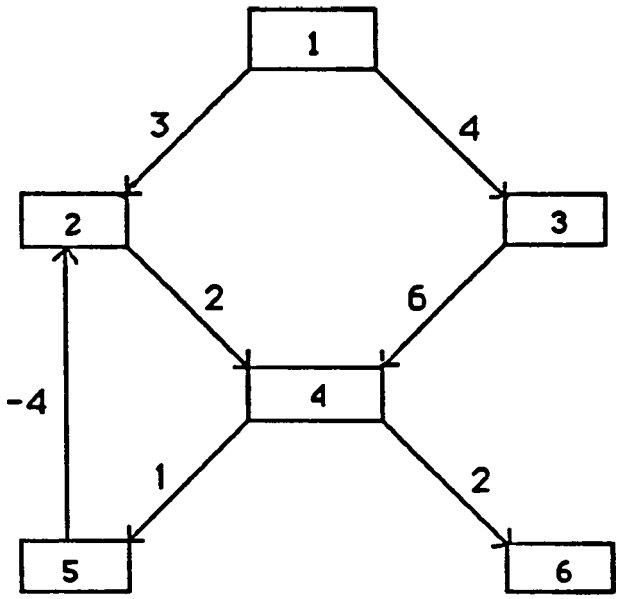

Figure 1. A weighted, directed graph with negative cycle [CHAN82].

III. IMPLEMENTATION

The fault tolerant version of the DSPA program is implemented in two layers corresponding to phases I and II of the computation. The first layer consists of the Primary version and an Alternate for each node of the graph. There is one Tester which controls the computation, sending the initialization values for each task and receiving the results. The computation is initiated at $node_1$, with each node in the system executing its primary version first. At the conclusion of computation which corresponds to the end of phase I, each of the nodes send its final result (obtained by the execution of the primary version) to the Consensus-Global Tester (Tester). The Tester verifies that the results are in compliance with the specifications. If no errors are found, the second phase of the computation is started. On the other hand, if the results are found to be erroneous, rollback occurs and recovery is initiated. These correspond to discarding the current values and invoking the alternate version.

When the Alternate at each node completes computation, it sends the final values to the Tester for validation. Once again compliance with the specifications is checked and if no errors are detected, the second phase is initiated. Otherwise the computation is aborted unless more alternate versions are available. A pictorial representation of the overall structure is shown in figure 2.

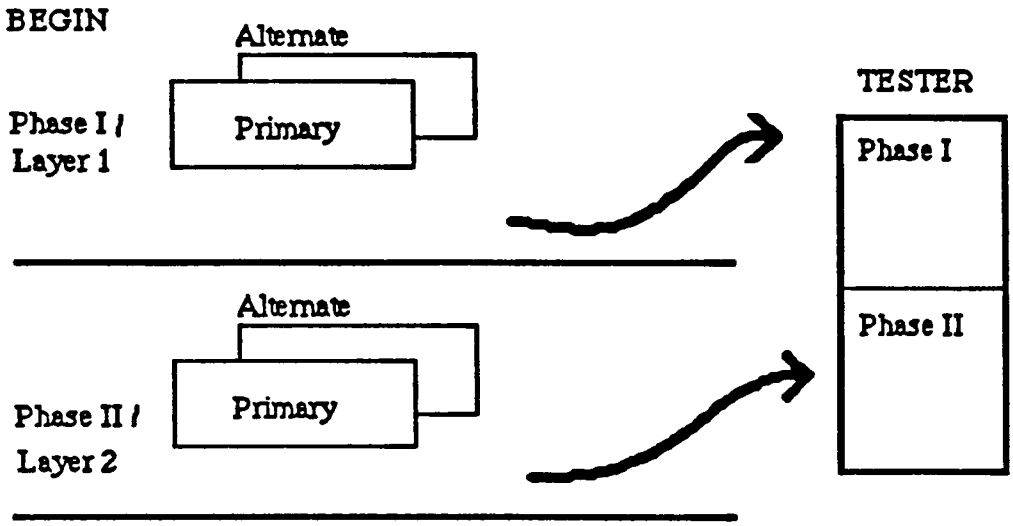

Figure 2. Overall structure of the implementation (Layer and Tester).

The second phase corresponding to layer 2 consists of two versions, a Primary and an Alternate. Computation is initiated at node 1, with each node executing its primary version for phase II. The primary version at the conclusion of its execution sends its results to the Tester for verification. If the results comply with the specifications provided, computation concludes. In the event of an error the alternate version for phase II is invoked and proceeds with the computation.

The Ada version is a procedure DSPA which consists of a task Tester to perform error detection. The first phase / layer consists of the procedures Layer1Primary, corresponding to the primary version and Layer1Second which is the Alternate. The second phase / layer is provided by procedures Layer2Primary

and Layer2Second which once again correspond to the Primary and alternate version for phase II. Each of these procedures consists of a collection of Ada tasks to perform the actual computation. An outline of the overall structure of procedure DSPA is as follows :

```
PROCEDURE DSPA IS

    ------------------------------------------

    TASK GlobalTester IS
    BEGIN
            -- Tester for layers
    END;

    ------------------------------------------

    PROCEDURE Layer1Primary IS
    BEGIN
            -- layer 1 primary module
    END;

    ------------------------------------------

    PROCEDURE Layer2Primary IS
    BEGIN
            -- layer 2 primary module
    END;

    ------------------------------------------

    PROCEDURE Layer1Second IS
```

```
        BEGIN
                -- layer 1 alternate module
        END;
        ------------------------------------------
        PROCEDURE Layer2Second IS
        BEGIN
                -- layer 2 alternate module
        END;
        ------------------------------------------
BEGIN
        -- GE = Global Error, NR = No Reply
        status := NR;
        Layer1Primary;
        IF status = GE THEN
                status := NR;
                Layer1Second;
        END IF;
        IF status = OK THEN
                status := NR;
                Layer2Primary;
                IF status = GE THEN
                        status := NR;
```

```
                    Layer2Second;
                END IF;
        END IF;
        IF status = GE THEN
                -- Error
        END IF:
END DSPA;
```

Figures 3 through 6 provide an overview of each of the layer procedures. A detailed explanation of the primary and alternate versions is provided in the following sections.

```
PROCEDURE Layer1Primary IS
        TASK TYPE Primary1 IS
        BEGIN
        END;
        TASK TYPE Primary IS
        BEGIN
        END;
L1P1 : Primary1;
L1P : ARRAY(2..N) OF Primary;
BEGIN
   LOOP  EXIT WHEN status /= NR;
  END LOOP;
END;
```

Figure 3. Layer 1 Primary modules overall structure.

```
PROCEDURE Layer2Primary IS
        TASK TYPE PrimaryII_1 IS
        BEGIN
        END;
        TASK TYPE PrimaryII IS
        BEGIN
        END;
L2P1 : PrimaryII_1;
L2P : ARRAY(2..N) OF PrimaryII;
BEGIN
        LOOP EXIT WHEN status /= NR;
        END LOOP;
END;
```

Figure 4. Layer 2 Primary modules overall structure.

```
PROCEDURE Layer1Second IS
        TASK TYPE CommunicationProcess IS
        BEGIN
        END;
        TASK TYPE Second1 IS
        BEGIN
        END;
        TASK TYPE Second  IS
        BEGIN
        END;
L1S1 : Second1;
L1S : ARRAY(2..N) OF Second;
CP : ARRAY(1..N) OF CommunicationProcess;
BEGIN
   LOOP  EXIT WHEN status /= NR;  END LOOP;
END;
```

Figure 5. Layer 1 Alternate modules overall structure.

```
PROCEDURE Layer2Second IS
        TASK TYPE CommunicationProcess IS
        BEGIN
        END;
        TASK TYPE SecondII_1 IS
        BEGIN
        END;
        TASK TYPE SecondII IS
        BEGIN
        END;
L2S1 : SecondII_1;
L2S : ARRAY(2..N) OF SecondII;
CP : ARRAY(1..N) OF CommunicationProcess;
BEGIN
  LOOP EXIT WHEN status /= NR; END LOOP;
END;
```

Figure 6. Layer 2 Alternate modules overall structure.

3.1 Primary Version

The primary version is implemented in two phases similar to the algorithm in [CHAN82]. Each phase consists of a procedure with nested tasks for each node of the graph. These are :

(1) Layer1Primary :: primary version for phase I / layer 1.

(a) Task L1P1 :: computation task for node1.

(b) Task L1P(i) :: computation task for nodes 2..N

(2) Layer2Primary :: primary version for phase II / layer 2.

(a) Task L2P1 :: computation task for node 1.

(b) Task L2P(i) :: computation tasks for nodes 2..N.

The task Tester initiates the overall computation by sending the initial values to each of the tasks. A task $L1P_i$ corresponding to node v_i implements phase I of the algorithm and computes the minimum distance. The shortest path computation is initiated by task L1P1 at node v_1 which sends length messages to its immediate successors and then loops, only accepting messages, until the number of outstanding acknowledgements becomes zero or a length message of less than 0 is received. At which time it sends a stop message to all its successors. The tasks L1P(i) for all other nodes accept and send messages until they receive the stop message. Each task upon receiving the stop message propagates it until all nodes receive such a message from each of its predecessors. Then all tasks send a copy of their final values for d, pred, num (path, predecessor and outstanding acknowledgements, respectively) to the Consensus-Global Tester (Tester) and completes execution.

The Tester accepts the results and performs a validity check based on the specifications it is provided. If no errors are found, the Tester sends the initialization values to task L2P1 (corresponding to phase II / layer 2 , node 1) and all other tasks. Phase II is then initiated by L2P1, which sends the appropriate over message. The contents of the initialization messages and the replies are given in the explanation of the Tester.

All primary tasks for phase I (L1P(i)) use three types of messages for communicating amongst themselves. The first, a length message, is triplet

(s,Pi,ack) where s is the path length, Pi the source address and ack the acknowledgement for previous length messages. The second is an entry call to entry point STOP, which is used to inform the nodes that phase I has ended. The third is an acknowledgement message (ack) used only to send acknowledgements to the task for node 1 (L1P1).

All tasks upon receiving a length message check whether the path length (s) is shorter than the current shortest path. If so, the tasks compute the values for propagating the message and then buffer them in the Table. The buffering of the shortest path continues until no more tasks are waiting for a rendezvous. At which time the new shortest path is propagated using length messages. If an even shorter path is subsequently received, it is written over the previous shortest path. The use of buffers ensures that only the most minimum of the length messages (of that particular round of messages) will be propagated and requires a buffer size of N - 1 in the worst case. Though a buffer of size N is convenient to declare and use.

During a rendezvous, tasks take the opportunity to return any acknowledgements which may still be owed to the calling task. This is achieved by the use of IN OUT parameters to exchange data. Thus while accepting a length message tasks also return acknowledgements which were buffered along with the previous length messages.

When the initialization message from the Tester is received, task L2P1 initiates the second phase by sending over- or over? messages to its successors. All Phase II tasks, L2P(i), use one type of message variable with two input parameters

consisting of the message type and the task id for communicating among themselves. A message value of 3 signifies an over-, whereas an over? is denoted by a message value of 4. The phase II tasks wait for the initialization message from the Tester and update the variables. Then each task waits for the initial message from a predecessor at which point it enters a loop which either accepts an over- or over? message, or propagates them. Computation for phase II tasks concludes when over messages from all successors have been received and propagated. At the end of phase II the values for d and over, corresponding to the shortest path and over message, are sent to the Tester for validation. In the event of an error, the Alternate for phase II is invoked under the assumption that phase I is correct. This can be safely assumed because the Tester (Consensus-Global) "passed" the phase I results.

The algorithm for the implementation follows. The code is in appendix B.

For process Primary1 PhaseI :

Accept Initialization values from tester

{loop}

If successor

    Send initial message

{end loop}

{ loop }

{ Select }

Accept length message

If s < 0 then computation done else acknowledge message

Or

Accept Ack message

decrement number of outstanding Acks

If number of Acks outstanding = 0 then computation done

Or

Accept stop

Update values

Or

When computation done

Send STOP to all successors

exit loop when all stops are processed

{ end loop }

Send values to Tester

For process Primary; PhaseI :

Accept Initialization values from tester

{ loop }

{ select }

```
Accept length message
   clear Ack
   If distance (s) < current shortest path (d)
         If num of Acks > 0 and predecessor = caller
            increment Ack count
         elsif number of acknowledgements > 0
            save Ack in buffer
         Update pred
         Update shortest path
         Save messages in buffer Table and increment Acks
         If outstanding Acks = 0
            add to ack count
   elsif distance (s) >= current shortest path (d)
         increment ack count
   Total all acks owed to calling task and clear buffer
Or
If no task waiting to rendezvous
      If any Ack message for process 1 in buffer
            Send it
      If any length message for process1
            Send it and receive acks
            Update
```

{loop}

If any message for successor

{select}

Send it and receive acks

Update

Or

delay

{end loop}

Or

Accept STOP

Update values

done = true

Or

When done

Send stop messages to successors

exit when all stop messages have been processed

{end loop}

Send values to tester

For process Primary1 PhaseII:

Accept Initialization values from tester

If distance (s) < 0

then message type = Over-

else message type = Over?

{ loop }

{ select }

Send Over message using delayed entry call until all are sent

Or

When tasks waiting to rendezvous

Accept message

Update values

Or

When all messages received and propagated, exit

{ end loop }

Send values to Tester

For process Primary; PhaseII :

Accept Initialization values from tester

Accept initial Over message and initialize message type

{ loop }

{ select }

When some task waiting to rendezvous

Accept message

Update values

Or

Send messages to all successors using delayed entry call

Or

When all messages received and propagated, exit

{ end loop }

Send values to Tester

3.2 Alternate Version

The alternate version, invoked in the event of an error by the primary, is implemented as two procedures corresponding to each Phase / Layer. Each procedure consists of three concurrently executing tasks for each vertex v_i of the graph. These are:

(1) Layer1Second :: alternate version for Phase I / Layer 1.

(a) Task L1S1 :: alternate for layer 1 node 1

(b) Task L1S(i) :: alternates for all other nodes in layer 1

(c) CP(i) :: communication / buffer process

(2) Layer2Second :: alternate version for phase II for Phase II / Layer 2.

(a) Task L2S1 :: alternate for layer 2 node 1

(b) Task L2S(i) :: alternates for all other nodes in layer 2

(c) CP(i) :: communication / buffer process

The Tester sends the initial values for each of the tasks, so that computation may be initiated. In the following, the notation $L1S_i$ will be used to denote all layer 1 alternate tasks and $L2S_i$ will correspond to layer 2 alternate tasks.

The task $L1S_i$ corresponding to node v_i implements phase I of the algorithm and computes the minimum distance. The tasks $L2S_i$ implement the second phase and ensure that all over messages are propagated. The computation is initiated by task L1S1 which is invoked when the procedure Layer1Second is called from DSPA.

Upon receiving the initialization values from the Tester, L2S1 sends length messages destined for its successors to its CP_1. The Communication Process (CP_1) in turn redirects them to the destination tasks. Each of the successors acts on the message accordingly. If the path received is shorter than the previous one, it is immediately propagated via CP. Otherwise an acknowledgement is sent to the calling task. The computation proceeds until task L1S1 receives either a path length less than 0 or its outstanding acknowledgements are 0. At which point, L1S1 (corresponding to node 1) sends a stop message to all its immediate successors, which are propagated to all tasks / nodes of the graph. Upon receiving a stop message each task propagates it. When all stop messages have been propagated, each task sends its final values for d, num and pred (path, outstanding

acknowledgements and predecessor) to the Tester. A pictorial representation of the relationship between processes and their CPs is given in figure 7 and the description of the initialization values is provided in the following section.

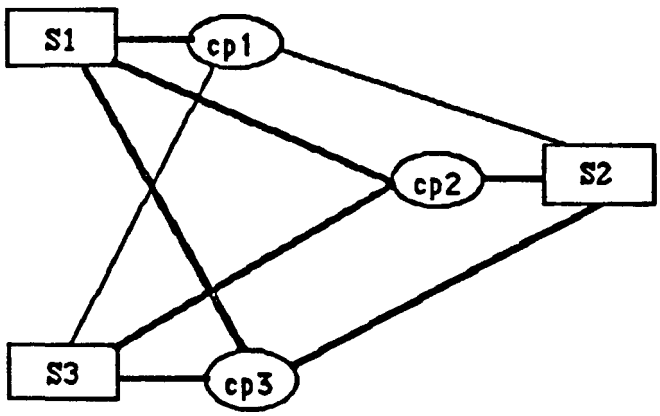

Figure 7. Relationship pathways for processes.

A message from any computation task to its corresponding CP is a 3-tuple (to,mtype,w) which provide the destination address, message type and path length. The CP which is used for phase II tasks also, can differentiate five types of computation messages depending on the parameter, mtype :

1 :: length message

2 :: acknowledgement message

3 :: over- message

4 :: over? message

5 :: stop message.

When a mtype 1 is received the CP redirects it to the destination as a length message 2-tuple (s,Pi) which are the path length and source address, respectively. Upon receiving a mtype 2, an acknowledgement message is sent to the destination. The message types 3 and 4 are used during phase II and correspond to an over- and over?. The final mtype i.e. 5 is sent as a stop message to indicate the termination of Phase I computation. It must be noted that all tasks communicate directly with the Tester, to receive the initialization values and send the final results, thus ensuring reliability.

The layer 2 tasks use one type of message to communicate with each other, that is :

over :: consists of two parameters, mtype and id. A value of 3 for mtype denotes an over- and 4 corresponds to an over?. The id corresponds to the task id.

Each phase II / layer 2 task receives its initialization message through the start message (from the Tester) and is then ready to compute, waiting for task L2S1 (phase II node 1) to initiate the computation. Each task propagates messages until all its successors are notified and then exits the processing loop. Subsequently sending its final values for d and the over message type to the Tester.

The testing philosophy, messages and interfaces with the Tester are exactly the same as for the Primary version. That is, upon the conclusion of the first phase all tasks send their results for verification to the Tester. After the Tester verifies the results the second phase is initiated. But, in case of an error another

Alternate may be invoked. Furthermore, messages and interfaces with the Tester are consistent. Consequently, no interface checks or changes need to be made.

In this particular case there are two versions for each phase, but we are not restricted to this. For example, a design similar to the alternate ($Second_i$) but with buffering of messages at the destination can be used as a third version. Another method to provide useful redundancy of design, is the use of different programmers.

3.3 Tester

The Consensus-Global Tester (Tester) is implemented as an Ada task and controls the computation by sending the initialization values to each task. It then receives the results from the computation tasks. When all tasks have responded by sending their final results, the Tester initiates its testing phase which ensures that all specifications are met. If an error is detected the Tester informs the procedure DSPA (using the variable status). Thus, the Alternate for that particular phase can be invoked. If all specifications are met the next phase is initiated or, if it is the last phase, computation successfully completes.

The initialization values for Phase I tasks are:

(1) A boolean list of successors.

(2) A list of lengths to the successors.

(3) The id number of each task (except the node 1 task).

(4) The number of predecessors.

The initialization values for Phase II tasks are:

(1) The shortest path.

(2) The number of outstanding acknowledgements.

(3) A list of successors.

(4) A list of predecessors.

(5) The id number of each task (except the node 1 task).

After ensuring that all the computation tasks have received their initialization values, the Tester waits at the Accept statement for the final values for d, pred, num (path length, predecessor and acknowledgements) from all the phase I computation tasks. It then performs the verification test and sets the variable status accordingly. A status of OK signifies that all tasks passed the test, whereas if status = GE (Global Error) the Alternate will have to be invoked. If a Global Error occurs, the Tester loops back to initialize the Alternates and thereafter waits, for messages from the alternate tasks.

The second part of the Tester uses a similar strategy to detect errors for phase II tasks. First initializing the tasks and subsequently waiting to receive the values, for the path length and the over message.

The inputs P_I for the Tester at end of phase I :

For all node i

receive $(d_i \wedge num_i \wedge pred_i \wedge id_i)$

The inputs P_{II} for the Tester at end of phase II :

For all node i

receive $(d_i \wedge over_i \wedge id_i)$

At the end of phase I, the Tester checks whether three assertions are met. These are: if the computation successfully concluded. Secondly, if any negative values for the shortest path are present. If so, either the predecessors shortest path must be negative or the path length must be negative. Finally, whether the shortest path (d) of a $node_i$ is equal to the shortest path of its predecessor plus the length from the predecessor to $node_i$.

(1) For $node_1$: num = 0 v s = 0.

(2) For $node_i$ (i=1..N) : $d_i < 0$ -----> ($d_{pred} < 0$ v $l_{pred,i} < 0$).

(3) For $node_i$ (i=1..N) : $d_i = d_{pred} + w_{pred,i}$.

At the end of phase II, the Tester checks whether the path length and over message correspond. That is:

(1) For $node_i$ (i=1..N) : $d_i < 0$ -----> over- message.

If the conditions are not met then an error condition is assumed and error recovery is initiated. The algorithm for the Tester follows.

<u>For process Tester</u>

{ loop }

```
Send Initialization values to all tasks
{ loop }
Accept message1 from all phase I tasks
{ end loop }
If specification met
   Update done and status
else if more alternates
   Update status
else
   Update status, done
exit when done
{ end loop }
If status = OK
   { loop }
    Send initialization values to all tasks
   { loop }
   Accept message 2 from phase II tasks
   { end loop }
    If specification met
        Update done and status
    else if more alternates
        Update status
```

```
else
   Update status, done
exit when done
{ end loop }
```

IV. ANALYSIS

In the following analysis the difficulties and problems alluded to in section II will be addressed within the context of the construct or methodology used to overcome them. Thus certain issues may be referred to several times, each will provide the technique, construct or methodology used to overcome the problem. The issue of reliability is treated separately.

4.1 Implementation Issues

The bidirectional inter-nodal communication inevitably leads to deadlock in distributed solutions, whereas centralized implementations are too restrictive and intolerant to faults. Reliance was placed on two techniques to overcome the problem of deadlock, these were: the use of an intermediary process (CP) in the alternate version and the use of Ada timed out entry call to provide nondeterminism on output in the primary. The intermediate / buffer process technique avoids deadlock by providing indirection. But poses two major drawbacks, in that, the implementation requires 2N concurrently executing processes for a graph of N nodes. Secondly, the number of messages also doubles. One message is required from task T_i to the corresponding CP_i and a second from CP_i to the task T_j. Though these drawbacks are associated with the use of intermediary processes, they stem from the constraints

placed by the language, which would not allow another deadlock free distributed implementation with such a high degree of parallelism.

Though the specifications of Ada do not explicitly provide for nondeterministic output, the use of timed out entry calls allow the mimicking of such nondeterminism. Non-determinism is the capability for a task to execute an alternate sequence of statements if the called task does not respond to a rendezvous. That is, it is not predetermined that a task will have to wait for its partner in a communication. It may execute alternate statements and at a later time, retry. The timed out entry call allows a sequence of statements to be executed alternatively if an entry call is not accepted within the specified duration. Thus

```
SELECT
   P1.message("entry call");
OR
  DELAY X;
  -- statements
END SELECT;
```

will execute statements following the DELAY, if P1 does not accept the call within X seconds. Consequently, tasks do not need to wait indefinitely for each other. It is worthy to note that the message passing is still synchronous i.e. the called task must respond. There is no message buffering capability. If a rendezvous is unsuccessful it can be attempted later. This allows two-way communication between tasks without resorting to the use of an intermediary process. Thus requiring only N tasks for a

graph of N nodes and a single message suffices for each communication attempt. The problem of deadlocking due to a circular wait situation is no longer an issue. The overhead of such a scheme is the delay incurred in waiting for a task, especially if the rendezvous is unsuccessful. Aside from the benefit of freedom from deadlock, a programmer can specify the time interval to wait for another task.

The asymmetric naming scheme in Ada was used to implement general entry points, which allowed tasks to be called without the server requiring prior knowledge of the callers identity. This allows greater flexability and generality in the implementation. But a drawback of this scheme is the lack of security it poses, as any task which knows the entry name can call and interfere with the server.

In Ada, communication is through an entry call made to the called task, which has a corresponding ACCEPT statement. The body of the ACCEPT statement acts as a critical section and no other communications can take place nor can any values be changed in the calling task until the conclusion of this rendezvous. The end of the rendezvous coincides with the end statement of the ACCEPT, in the called task. If the ACCEPT has no body or parameters it acts as a synchronization primitive only and no information is exchanged. The use of parameters allows information to be exchanged by reference or value. The benefit of such a two-way scheme is the ability to exchange length messages and acknowledgements in the same communication. Thus circumventing the need for a task to explicitly send acknowledgements to its predecessors. This was effectively shown in the primary version of the implementation, which buffered acknowledgements until the particular

task called with another length message. At which time the acknowledgements were exchanged with the length message. The obvious drawback of this scheme is that acknowledgements are always delayed until the predecessor attempts to communicate. Thus predecessor tasks are always a little "behind" in the information they possess. This is especially true if the task owed the acknowledgements does not communicate again and when computation concludes, the number of outstanding acknowledgements may have an effect on the eventual outcome.

The number of messages propagated in the Alternate implementation is very large. In the worst case N (N - 1 messages + 1 EOT) messages are sent from a task to its CP and the CP propagates N - 1 of those, thus approximately $2N^2$ messages are used for N nodes. The number of messages is large not only for the reason stated above, but also because length messages are propagated even though a following message may provide a shorter path. In the Primary version the use of ATTRIBUTES indirectly provides the capability to reduce the number of messages. The syntax is, P'COUNT, which provides the number of tasks waiting at entry point P. It allows the implementation of priorities at a very crude level. Thus tasks can prioritize messages, with in-coming messages having first preference. Outgoing messages are buffered until no tasks are waiting to rendezvous. That is, P'COUNT is equal to 0. This ensures that the shortest path will be propagated after a round of messages and the others will be discarded. In the primary version use of the COUNT Attribute coupled with the modification to buffer messages was instrumental in reducing the number of length messages which are propagated.

Considering, that in the worst case, the primary propagates N^2 messages (N nodes each sending N-1 messages) any reduction is a help.

The use of the Communication / Buffer Process scheme provides an arbitrarily greater degree of concurrency when compared to the nondeterministic (Primary version) message sending. Since tasks, using timed out entry calls, need to delay for a message to get through they are unable to do any thing else. Whereas the CP (Alternate) version sends its messages and can then continue processing. It essentially frees up the task to do something else. In the Primary, the task must itself wait and synchronize with the called task.

The Attribute CALLABLE which returns true if a task is not aborted, terminated or in an abnormal state, was used to aid in message sending. It essentially provided the capability to check a tasks ability to accept messages. Though care must be taken in its use, as a task may infact terminate between the time of the check and the actual message.

In each case it is clear that the language put constraints on the programmer to implement the algorithm, but in each case the flexibility of the language was used to achieve the objective. Though this was achieved by moulding the language to fit the needs. Examples are, the use of timed out entry calls to achieve nondeterminism for output messages. Secondly the use of the Attribute to implement priorities, however crudely. Both these capabilities were major factors in providing efficiency and deadlock freedom for the implementation.

4.2 Reliability Issues

Distributed programs as indicated above are difficult to implement and in contrast to sequential programs require the satisfaction of both the safety and liveness properties. Therefore requiring care in the implementation. But this still does not guarantee a correct solution, thus fault tolerant techniques are required to provide some measure of reliability. This reliability can be achieved through careful structuring and design of the program and the use of error detection and recovery techniques. In the implementation of the Distributed Shortest Path Algorithm, the concept of Communication-Closed Layers was used to provide Safe Layers. This was then extended by the use of a Consensus-Global Tester to provide error detection and recovery capabilities.

The unrestricted communication and the interactive nature of distributed programs make them difficult to verify formally. The use of fault prevention techniques, such as testing, reduce errors but residual design inadequacies may still be present. One method to design programs and provide fault tolerance is the technique of safe layering. As described previously, the objective is to partition concurrent programs into concurrently executing segments and to allow communication only within the layers thus created. The Distributed Shortest Path Algorithm by its nature provided an extremely good opportunity to partition it into two layers, corresponding to the two phases of the computation. The logical separation was extended to the physical program, with the provision of two versions

for each phase. Error detection is provided through the use of Consensus-Global Testers.

The major strength of he DSPA program is its ability to continue processing even in the event of faults. The reliability inherent in fault tolerant software is based on useful redundancy. If the Primary fails then a more inefficient module, or one which provides degraded results, will be executed. Each successive alternate version provides continued service but at a degraded level of efficiency or output. The handling of the faults, rollback and recovery, are transparent to the user.

In the DSPA implementation it should be noted that the Primary requires N + 1 concurrently executing tasks (N nodes + Tester) and approximately N^2 messages have to be propagated in the worst case. Whereas the Alternate consists of 2N + 1 concurrent processes and requires $2N^2$ messages to be propagated. Thus the alternate version would be costlier in terms of the overhead for processes and the number of messages propagated. But fault tolerant applications themselves, are inherently more inefficient than non-fault tolerant ones.

In the case of the DSPA program, the overhead comes from the extra number of messages required to communicate with the Tester. The initialization for each phase requires N messages and N replies are sent at the conclusion. If an error is detected the Alternate needs to be initialized, thus N more messages are sent. Consequently, in the worst case 8N messages would be needed and in the best case 4N. This does not take into account the overhead of N messages (minimal) for the stop messages during phase I. But the advantages of fault tolerance are far greater

than the drawbacks. For example, fault tolerant software provides continued service, even in the event of faults. In the DSPA program continued service can be provided through the use of the Alternate, which will be invoked in the event that the Primary fails to meet its specifications.

The reliability provided by fault tolerant software is based on the concept of useful redundancy. That is the provision of spares with different algorithms, written in different languages or by different programmers. The premise is that residual design inadequacies of one will not be present in another. Thus based on the concept of useful redundancy two modules were provided in the DSPA program. To further ensure that the programs would be correct, the Safe Layering technique was used to partition the program in two Communication-Closed Layers. An advantage of this technique is that, errors caught, lead to a rollback of only that particular layer. Thus valuable time is not lost reinitiating the entire computation. A second advantage is that errors are caught as early as possible in the computation. That is, a fine partitioning allows errors to be detected at the earliest. In the DSPA case, an error detected in phase 2 need only cause a rollback to the begining of layer 2. Secondly, if a fault occurs in phase I it is detected prior to the initiation of phase II. Without Safe Layering the error would be detected at the conclusion of computation, when the test would be performed.

The use of fault tolerant techniques and the provision of fault tolerance in software provides reliability but at an increased cost, in terms of the messages. But the overhead is minimal compared to the provision of continued service, reliability

and the ability to design safe programs, detect errors and correct them. It should be noted that the analysis concerning message overheads takes the propagation of a length message into account, not the overall computation.

V. CONCLUSION

It is apparent that distributed algorithms are difficult to implement and are affected by the constraints inherent in the constructs for concurrency provided by Ada. Situations leading to deadlock are pervasive as communication is unrestricted. Whereas, attempts to solve the deadlock problem have tremendous overheads in terms of the number of messages required and the number of processes running concurrently. In addition to the deadlock problem, issues such as memory usage, amount of concurrency and the number of processes executing simultaneously have to be addressed. The solutions to these problems are not easy to find. This puts the burden on the programmer who, as the complexity of the algorithm increases, is more likely to make errors in converting the algorithm to code. His choices will ultimately affect the overall outcome. Incorrect choices may have adverse effects, not only decreasing performance but ultimately leading to problems. Such problems are hard to detect and harder to correct within the confines of the language, especially when trying to maintain a high level of concurrent activity.

The quality and elegance of a solution depends to a large extent on the programmer ability to forsee problems and solve them through judicious use of the language constructs. Though concurrency in the algorithm and concurrent constructs in the programming language are helpful and effect performance, the choice of the language is very important in achieving maximal performance. The flexibility of the language plays an important role in allowing solutions without incurring

unacceptable overhead, in terms of the number of processes or the number of messages passed. The ability of the language to provide tools, either directly or indirectly, is useful. Such an ability may be utilized by a programmer in cases where regular constructs are too confining or inadequate. But features useful to a programmer should be easily available. He should not have to resort to the use of constructs to mold the language into a shape which allows algorithms to be implemented.

In the case of distributed algorithms where the problem of deadlock looms large we can either depend on the programmer and the languages flexibility, or use indirection (as shown by the use of CP in the Alternate implementation). The drawbacks of both are evident. Two methods which would serve better are the extension of the language to provide the necessary features, or deadlock detection followed by arbitration. Since the latter is considerably harder to achieve, a language must provide either the capability to buffer messages implicitly or the ability to send messages nondeterministically. In the event of an inability to implement a deadlock free solution within the constraints of a language, the use of the Indirection methodology is suggested. That is, the use of CP type tasks to ensure freedom from deadlock. This technique will be invaluable in providing a quick and easy solution, while a more elegant one is thought out.

Distributed programs by virtue of their complexity are extremely difficult to verify formally. This is due to the unrestricted communication between interacting processes with unpredictable orders of execution. Thus fault prevention methods are

insufficient and reliance must be placed on software fault tolerance, under the assumption that residual design inadequacies are present and may mainfest themselves at some later time.

The distributed program must be designed using a methodology to decrease the chances of a breach of the correctness property and error detection and recoverability must be provided. If an error is detected the alternate modules can be invoked to provide uninterrupted service. The lack of ability to verify programs should not be used as an excuse, in the event of an error. Since techniques exists that can provide reliability even in the event of errors.

The use of fault tolerant techniques have certain drawbacks, specifically the overhead for messages and the maintenance of extra versions. But the benefits, in terms of the reliability they provide, far outweigh the drawbacks.

The field of distributed computing is still in its infant stages and the study of the implementation aspects of distributed algorithms within constraints placed by current languages will prove invaluable in the future. Similarly new techniques and methodologies are needed in fault tolerance to ensure better error detection and recovery capabilities.

APPENDIX A

DISTRIBUTED SHORTEST PATH ALGORITHM

Phase I for Process Pj J ≠ 1

```
begin    d := ∞; pred is Undefined; num := 0 end;
{ Upon receiving a length message (s,Pi) }
If s < d then begin
   { send ack to pred, the prefinal vertex on previous shortest path }
   If num > 0 then
         send an ack to pred;
   { update pred, d }
   pred := Pi;
      d := s;
   { send len message to all successors and increment num accordingly, then
       return ack to pred if num = 0 }
   Send a len message (d+w,Pj) to all successors;
   num := num + number of successors;
   If num = 0 then
         Send ack to pred
end
```

```
else { s >= d }
   Send ack to Pi;
{ Upon receiving an ack }
begin
   {decrement number of unacknowledged messages }
   num := num - 1;
   { send acknowledgement to pred if acks have been received for all
        messages }
   If num = 0 then
          Send ack to pred
end;
```

Phase I for process P1

```
d := 0; pred is Undefined;
Send (w,P1) to all successors; num := number of successors;
{upon receipt of a length message }
{ start Phase II if negative cycle detected }
If s < 0 then
      terminate Phase I and start Phase II;
else
```

return ack to Pi;

{ Upon receiving ack}

{ update num; start phase II if there is no unacknowledged message remaining }

num := num - 1;

If num = 0 then

terminate Phase I and start Phase II.

Phase II for process Pj (j ≠ 1) with num = 0

{ Upon receiving an over- message }

If d ≠ -∞ then begin

d := -∞;

Send over- to all successors;

end;

{Upon receiving an over? message }

If d ≠ -∞ then

Send over? to all successors who have not been sent such a message;

Phase II for process Pj (j ≠ 1) with num > 0

{ Upon receiving a Phase II message (over- or over?) }

If $d \neq -\infty$ then begin

    d := -∞;

    Send over- to all successors;

  end;

Phase II for process P1

If P1receives a message (s,Pi) with $s < 0$ during Phase I then

    { P1 detects a negative cycle }

  Send an over- message to all successors

else { num = 0 for P1 at the end of Phase I }

  Send over? message to all successors;

APPENDIX B

ADA CODE

```
PROCEDURE DSPA IS
TYPE statustype IS (OK,NR,GE);
TYPE item1 IS ARRAY(1..N,1..N) OF BOOLEAN;
TYPE item2 IS ARRAY(1..N,1..N) OF INTEGER;
TYPE item3 IS ARRAY(1..N) OF BOOLEAN;
TYPE item4 IS ARRAY(1..N) OF INTEGER;

-------------------------------------------------------------------------------
TASK TYPE GlobalTester IS
     ENTRY message0(s1,d,pred,num : IN INTEGER);
     ENTRY message1(d,pred,num ,j: IN INTEGER);
     ENTRY message2(d,over,j: IN INTEGER);
END;
TASK BODY GlobalTester IS
     more : INTEGER;
     done : BOOLEAN;
     error : BOOLEAN;
     successor : item1;
     predecessor : item1;
     s : INTEGER;
     w : item2;
     j : INTEGER;
     pcount : ARRAY(1..N) OF INTEGER;
     phase1array : ARRAY(1..N,1..3) OF INTEGER;
     phase2array : ARRAY(1..N,1..2) OF INTEGER;
     ---------------------------------------------------------------------------
     FUNCTION assertion1 RETURN BOOLEAN IS
     i : INTEGER := 2;
     error : BOOLEAN := false;
     BEGIN
         IF s  /= 0 AND phase1array(1,3) /= 0 THEN
              error := true;
         END IF;
         WHILE NOT error AND i <= N LOOP
              IF phase1array(i,1) /= (phase1array(phase1array(i,2),1) +
                        w(phase1array(i,2),i)) THEN
                   error := true;
              END IF;
              IF phase1array(i,1) < 0 THEN
                   IF phase1array( phase1array(i,2),1) >= 0 AND
```

```
                w( phase1array(i,2),i) >= 0 THEN
                error := true;
            END IF;
        END IF;
        i := i + 1;
    END LOOP;
    RETURN error;
END assertion;
-----------------------------------------------------------------------------
FUNCTION assertion2 RETURN BOOLEAN IS
error : BOOLEAN := false;
j : INTEGER := 0;
k : INTEGER := 0;
BEGIN
    FOR i IN 1..N LOOP
        IF phase2array(i,1) < 0 AND THEN phase2array(i,2) /= 3 THEN
            error := true;
        END IF;
        EXIT WHEN error;
    END LOOP;
    RETURN error
END assertion2;
-----------------------------------------------------------------------------
BEGIN
    -- read values for all variables from file
    more := 1;
    done := false;
    error := false;
    LOOP
        DELAY 3;
        IF more = 1 THEN
            L1P1.start(successor(1,1..N),w(1,1..N),pcount(1));
            FOR i IN 2..N LOOP
                j := i;
                L1P(i).start(successor(i,1..N),w(i,1..N),j,pcount(i));
            END LOOP;
        ELSE
            L1S1.start(successor(1,1..N),w(1,1..N),pcount(1));
            FOR i IN 2..N LOOP
                j := i;
                L1S(i).start(successor(i,1..N),w(i,1..N),j,pcount(i));
            END LOOP;
        END IF;
        ACCEPT message0(s1,d,pred,num : IN INTEGER) DO
```

```
            phase1array(1,1) := d;
            phase1array(1,2) := pred;
            phase1array(1,3) := num;
            s := s1;
        END;
        FOR i IN 2..N LOOP
            ACCEPT message1(d,pred,num,j : IN INTEGER) DO
                phase1array(j,1) := d;
                phase1array(j,2) := pred;
                phase1array(j,3) := num;
            END;
        END LOOP;
        IF assertion1 THEN
            done := true;
            status := OK;
        ELSIF more < 2 THEN
            more := more + 1;
            status := GE;
        ELSE
            status := GE;
            done := true;
        END IF;
        EXIT WHEN done OR status = OK;
    END LOOP;
    IF status = OK THEN
        done := false;
        more := 1;
        LOOP
            DELAY 3;
            IF more = 1 THEN
                L2P1.start(phase1array(1,1),phase1array(1,3),
                                successor(1,1..N),predecessor(1,1..N));
                FOR i IN 2..N LOOP
                    j := i;
                    L2P(i).start(phase1array(i,1),phase1array(i,3),
                                successor(i,1..N, predecessor(i,1..N) ,j);
                END LOOP;
            ELSE
                L2S1.start(phase1array(1,1),phase1array(1,3),
                            successor(1,1..N), predecessor(1,1..N));
                FOR i IN 2..N LOOP
                    j := i;
                    L2S(i).start (phase1array(i,1),phase1array(i,3),j
                                successor(i,1..N), predecessor(i,1..N) );
```

```
            END LOOP;
        END IF;
        FOR i IN 1..N LOOP
            ACCEPT message2(d,over ,j: IN INTEGER) DO
                phase2array(j,1) := d;
                phase2array(j,2) := over;
            END;
        END LOOP;
        IF assertion2 THEN
            done := true;
            status := OK;
        ELSIF more < 2 THEN
            more := more + 1;
            status := GE;
        ELSE
            status := GE;
            done := true;
        END IF;
        EXIT WHEN done OR status = OK;
    END LOOP;
  END IF;
END GlobalTester;
-------------------------------------------------------------------------------
PROCEDURE Layer1Primary( Tester : IN OUT GlobalTester) IS
    ---------------------------------------------------------------------------
    TASK TYPE Primary1 IS
        ENTRY len_msg(s1,Pi : IN INTEGER;ack : OUT INTEGER);
        ENTRY get_ack(ack : IN INTEGER);
        ENTRY start(ary1:item3,ary2 : item4;pcount : IN INTEGER);
        ENTRY stop;
    END;
    -- Phase I    task Primary1
    TASK BODY Primary1 IS
            d : INTEGER := 0;                -- current shortest path
            num : INTEGER := 0;              -- unacknowledged messages
            pred : INTEGER := 0;         -- most current predecessor
            XXXXX : INTEGER;
            count : INTEGER;
            successor : item3;
            ack : INTEGER;
            s : INTEGER := 0;
            w : item4;          -- weights to successors
            done : BOOLEAN := FALSE;        -- true if computation done
            i : INTEGER := 0;
```

```
        ----------------------------------------------------------------------
        FUNCTION moretodo RETURN BOOLEAN IS
        done : BOOLEAN := false;
        i : integer := 1;
        BEGIN
            WHILE NOT done AND i <= N LOOP
                IF successor(i) THEN
                    done := true;
                END IF;
                i := i + 1;
            END LOOP;
            RETURN done;
        END moretodo;
        ----------------------------------------------------------------------
    BEGIN
        ACCEPT start(ary1 : item3;ary2 : item4;pcount : INTEGER) DO
            successor := ary1;
            count := pcount;
            w := ary2;
        END;
        -- send initial length messages to successors
        FOR i IN 2..N LOOP
            IF successor(i) THEN
                L1P(i).len_msg(w(i),1,ack);
                num := num + 1;
            END IF;
        END LOOP;

        LOOP
            SELECT
                -- accept length mesg. If path < 0 then computation done
                -- else acknowledge mesg
                ACCEPT len_msg(s1,Pi : IN INTEGER;ack : OUT INTEGER)
DO
                        IF s1 < 0 THEN
                            done := TRUE;
                            s := s1;
                        ELSE
                            ack := 1;
                        END IF;
                    END;
            OR
                -- accept ack mesg and decrement outstanding acks.
```

```
            -- If no acks remain then done
            ACCEPT get_ack(ack : IN INTEGER) DO
                num := num - ack;
                IF num = 0 THEN done := TRUE; END IF;
            END;
        OR
            ACCEPT stop DO
                    count := count - 1;
            END;
        OR
            WHEN done =>
                FOR i IN 2..N LOOP
                    IF successor(i) THEN
                        SELECT
                            L1P(i).stop;
                            successor(i) := false;
                        OR
                            DELAY XXXXX;
                            NULL;
                        END SELECT;
                    END IF;
                END LOOP;
                IF count = 0 AND NOT moretodo THEN
                    EXIT
                END IF;
        END SELECT;
    END LOOP;
    Tester.message0(s,d,pred,num);
END Primary1;

-------------------------------------------------------------------------------
    TASK TYPE Primary IS
        ENTRY len_msg(s,Pi : IN INTEGER;ack : OUT INTEGER);
        ENTRY stop;
        ENTRYstart(ary1:item3;ary2 : item4;id : INTEGER;pcount : INTEGER);
    END;
    -- Phase I    task Primary_j    (j:2..N)
    TASK BODY Primary IS
            d : INTEGER := INTEGER'LAST;    -- current shortest path
            num : INTEGER := 0;  -- unacknowledged messages
            pred : INTEGER := 0;  -- most current predecessor
            self : INTEGER;
            count : INTEGER;
            numct : INTEGER := 0;
```

```
            ack : INTEGER;
            XXXXX : INTEGER;  -- time to delay
            successor : item3;
            w : item4;
            Table : ARRAY(1..N,1..3) OF INTEGER;
             -- col. 1=weight ; col. 2=acks ; col. 3=if mesg to be sent
        ------------------------------------------------------------------------
        FUNCTION moretodo RETURN BOOLEAN IS
        done : BOOLEAN := false;
        i : integer := 1;
        BEGIN
            WHILE NOT done AND i <= N LOOP
                IF successor(i) THEN
                    done := true;
                END IF;
                i := i + 1;
            END LOOP;
            RETURN done;
        END moretodo;
        ------------------------------------------------------------------------
    BEGIN
        ACCEPT start(ary1 : item3;ary2:item4;id:INTEGER;pcount :INTEGER)
DO
            successor := ary1;
            self := id;
            count := pcount;
            w := ary2;
        END;
        -- initialize self , Table to  0 , 0 and -1
        FOR i IN 1..N LOOP
            Table(i,1) := 0; Table(i,2) := 0; Table(i,3) := -1;
        END LOOP;
        LOOP
            SELECT
                -- accept length message
                ACCEPT len_msg(s,Pi : IN INTEGER;ack : OUT INTEGER)
DO
                    ack := 0;
                    -- if path is shorter than previous and old predecessor is
                    -- same as current caller then acknowledge else save ack in
table
                    IF s < d THEN
                            IF numct > 0 AND pred = Pi THEN
                            ack := ack + 1;
```

```
                    ELSIF numct > 0 THEN
                        Table(pred,2) := Table(pred,2) + 1;
                    END IF;
                        -- update predecessor and current shortest path
                        pred := Pi;
                        d := s;
                        -- buffer shortest path to successor
                        FOR i IN 1..N LOOP
                        IF successor(i) THEN
                            Table(i,1) := d + w(i);
                            Table(i,3) := i;
                            numct := numct + 1;
                        END IF;
                       END LOOP;
                       -- if no outstanding acks then acknowledge
                    IF numct = 0 THEN ack := ack + 1; END IF;
                    -- if path is longer than previous shortest path
                    -- then acknowledge/save
                    ELSE IF s >= d THEN
                        ack := ack + 1;
                    END IF;
                    ack := ack + Table(pi,2);
                    Table(pi,2) := 0;
    END;
OR
    -- no one waiting to rendezvous
    WHEN len_msg'COUNT = 0 =>
        -- if there is an ack for P1 then send it
        IF Table(1,2) > 0 THEN
            SELECT
                L1P1.get_ack(Table(1,2));
                Table(1,2) := 0;
            OR
                DELAY XXXXX;
                NULL;
            END SELECT;
        END IF;
        -- if P1 is successor then send length mesg and get acks
back
        IF successor(1) AND Table(1,3) /= -1 THEN
            L1P1.len_msg(Table(1,1),self,ack);
            num := num + 1 - ack;
            Table(1,3) := -1;
        END IF;
```

```
                -- for all other tasks, if they are successors then send length
                -- message. But wait xxxxxx seconds only for a rendezvous
                FOR i IN 2..N LOOP
                  IF successor(i) AND Table(i,3) /= -1 THEN
                        SELECT
                           L1P(i).len_msg(Table(i,1),self,ack);
                           num := num + 1 - ack;
                           Table(i,3) := -1;
                        OR
                           DELAY XXXXX;
                           NULL;
                        END SELECT;
                  END IF;
                END LOOP;
                numct := num;
            OR
                ACCEPT stop DO
                    count := count - 1;
                    done := true;
                END;
            OR
                WHEN done =>
                    FOR i IN 2..N LOOP
                        IF successor(i) THEN
                            SELECT
                                L1P(i).stop;
                                successor(i) := false;
                            OR
                                DELAY XXXXX;
                                NULL;
                            END SELECT;
                        END IF;
                    END LOOP;
                    IF count = 0 AND NOT moretodo THEN
                        EXIT;
                    END IF;
            END SELECT;
        END LOOP;
        Tester.message1(d,pred,num,self);
    END Primary;
    ------------------------------------------------------------------------------
L1P1 : Primary1;
L1P : ARRAY (2..N) OF Primary;
BEGIN
```

```
    LOOP
        EXIT WHEN status /= NR;
    END LOOP;
END Layer1Primary;

--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
PROCEDURE Layer2Primary(Tester : IN OUT GlobalTester) IS
    ----------------------------------------------------------------------------
    TASK TYPE PrimaryII_1 IS
        ENTRY over(mtype : IN INTEGER;id : INTEGER);
        ENTRY start(s1,num1 : IN INTEGER;ary1 , predecessor : item3);
    END;
    -- phase II    task P1
    TASK BODY PrimaryII_1 IS
            msg : INTEGER;    -- message to be sent
            num : INTEGER;    -- unacknowledged messages
            s : INTEGER;      -- distance received
            successor : item3;
            predary : item3;
            self : INTEGER;
            XXXXX : INTEGER;  -- time to delay
            backup : item3;
            change : BOOLEAN := true;
        ------------------------------------------------------------------------
        FUNCTION moretodo(ary : item3) RETURN BOOLEAN IS
        done := BOOLEAN := false;
        i : integer := 1;
        BEGIN
            WHILE NOT done AND i <= N LOOP
                IF ary(i) THEN
                    done := true;
                END IF;
                i := i + 1;
            END LOOP;
            RETURN done;
        END moretodo;
        ------------------------------------------------------------------------
    BEGIN
        END;
        ACCEPT start(s1,num1 : IN INTEGER;ary1,predecessor:item3) DO
            successor := ary1;
            predary := predecessor;
```

```
            backup := successor;
            s := s1;
            num := num1;
        END;
        IF s < 0 THEN
            msg := 3;
        ELSE
            msg := 4;
        END IF;
        LOOP
            SELECT
                IF change THEN
                    FOR i IN 2..N LOOP
                        -- for all successors send over message.
                        -- But wait only xxxxx seconds for
                        -- a rendezvous.
                        IF successor(i) THEN
                            SELECT
                                IF L2P(i)'CALLABLE THEN
                                    L2P(i).over(msg,1);
                                END IF;
                                successor(i) := FALSE;

                            OR
                                DELAY XXXXX;
                                NULL;
                            END SELECT;
                        END IF;
                    END LOOP;
                    IF NOT moretodo(successor) THEN change := false; END
IF;
                END IF;
            OR
                WHEN over'COUNT > 0 =>
                    ACCEPT over(mtype, id: IN INTEGER) DO
                        predary(id) := false;
                        IF msg = 4 AND mtype = 3 THEN
                            msg := 3; d := INTEGER'FIRST;
                            change := true;
                            successor := backup;
                        END IF;
                        msg := mtype;
                    END;
            OR
```

```
                IF NOT change AND NOT moretodo(predecessor) THEN
                    EXIT;
                END IF;
                END SELECT;
            END LOOP;
            Tester.message2(s,msg,1);
        END PrimaryII_1;
        ------------------------------------------------------------------------------

        TASK TYPE PrimaryII IS
            ENTRY over(mtype , id : IN INTEGER);
            ENTRY start(d1,num1 : IN INTEGER;ary1,predecessor : item3; id :
INTEGER);
        END;
        -- Phase II      task Pj     (j:2..N)
        TASK BODY PrimaryII IS
            num : INTEGER;  -- unacknowledged messages
            d : INTEGER;    -- current shortest path
            msg : INTEGER;      -- message received and sent later
            successor : item3;
            backup : item3;
            predary : item3;
            self : INTEGER;
            XXXXX : INTEGER;  -- time to delay
            change : BOOLEAN := true;
            ------------------------------------------------------------------
            FUNCTION moretodo(ary : item3)  RETURN BOOLEAN IS
            done : BOOLEAN := false;
            i : integer := 1;
            BEGIN
                WHILE NOT done AND i <= N LOOP
                    IF ary(i) THEN
                        done := true;
                    END IF;
                    i := i + 1;
                END LOOP;
                RETURN done;
            END moretodo;
            ------------------------------------------------------------------
        BEGIN
            ACCEPT start(d1,num1 : IN INTEGER;ary1,predecessor : item3;id :
INTEGER ) DO
                d := d1;
                self := id;
```

```
        predary := predecessor;
        successor := ary1;
        backup := successor;
        num := num1;
    END;
    -- accept over message from predecessor
    ACCEPT over(mtype , id: IN INTEGER) DO
        msg := mtype;
        IF num > 0 AND msg /= 3 THEN
            msg :=  3;
        END IF;
        predary(id) := false;
        IF msg = 3 THEN d := INTEGER'FIRST; END IF;
    END;
    LOOP
        SELECT
            -- if some task is waiting to rendezvous then accept its attempt
            WHEN over'COUNT > 0 =>
                ACCEPT over(mtype , id: IN INTEGER) DO
                    IF msg = 4 AND mtype = 3 THEN
                        msg := 3; d := INTEGER'FIRST;
                        change := true;
                        successor := backup;
                    END IF;
                    predary(id) := false;
                END;
        OR
            IF change THEN
                FOR i IN 2..N LOOP
                    -- for all successors, send over message.
                    -- But wait only xxxxx seconds
                    -- for a rendezvous
                    IF successor(i) THEN
                            SELECT
                                IF  L2P(i) ' CALLABLE THEN
                                      L2P(i).over(msg,self);
                                END IF;
                                successor(i) := FALSE;
                            OR
                                 DELAY XXXXX;
                                  NULL;
                            END SELECT;
                       END IF;
                END LOOP;
```

```
                        IF NOT moretodo(successor) THEN
                            change := false;
                        END IF;
                    END IF;
                OR
                    IF NOT moretodo(successor) AND NOT moretodo(predary)
THEN EXIT END IF;
                END SELECT;
            END LOOP;
            Tester.message2(d,msg,self);
        END PrimaryII;
        ------------------------------------------------------------------------------------
    L2P1 : PrimaryII_1;
    L2P : ARRAY (2..N) OF PrimaryII;
    BEGIN
        LOOP
            EXIT WHEN status /= NR;
        END LOOP;
    END Layer2Primary;

------------------------------------------------------------------------------------

------------------------------------------------------------------------------------
    PROCEDURE Layer1Second(Tester : IN OUT GlobalTester) IS

------------------------------------------------------------------------------------
        TASK CommunicationProcess IS
            ENTRY msg(to1,mtype1,w1 : IN INTEGER);
            ENTRY idself(id : IN INTEGER);
        END;
        TASK BODY CommunicationProcess IS
            self : INTEGER;
            tctr : INTEGER;
            to : INTEGER;
            i : INTEGER;
            Table : ARRAY(1..N,1..3) OF INTEGER;
            ------------------------------------------------------------------------------
            PROCEDURE compact IS
                i : INTEGER;
                j : INTEGER;
                swap : ARRAY(1..N,1..3) OF INTEGER;
            BEGIN
                j := 1;
                FOR i IN 1..N LOOP
```

```
            IF Table(i,1) /= -1 THEN
                swap(j,1) := Table(i,1);
                swap(j,2) := Table(i,2);
                swap(j,3) := Table(i,3);
                j := j + 1;
            END IF;
        END LOOP;
        swap(j,1) := -1;
        i := 1; j := 1;
        WHILE swap(j,1) /= -1 LOOP
            Table(i,1) := swap(j,1);
            Table(i,2) := swap(j,2);
            Table(i,3) := swap(j,3);
            i := i + 1; j := j + 1;
        END LOOP;
        Table(i,1) := -1;
        tctr := i;
    END compact;

-------------------------------------------------------------------------------
    BEGIN
        -- initialize Table to -1 and self
        LOOP FOR i IN 1..N LOOP
            Table(i,1) := -1; Table(i,2) := -1;
            Table(i,3) := -1;
        END LOOP;
        ACCEPT idself(id : INTEGER) DO
            self := id;
        END;
        tctr := 1;
        LOOP
            SELECT
                ACCEPT msg(to1,mtype1,w1: IN INTEGER) DO
                    to := to1;
                    Table(tctr,1) := to1;
                    Table(tctr,2) := mtype1;
                    Table(tctr,3) := w1;
                    tctr := tctr + 1;
                END;
                LOOP
                    EXIT WHEN to = -1;
                    ACCEPT msg(to1,mtype1,w1: IN INTEGER) DO
                        to := to1;
                        Table(tctr,1) := to1;
```

```
                    Table(tctr,2) := mtype1;
                    Table(tctr,3) := w1;
                    tctr := tctr + 1;
                END;
            END LOOP;
        OR
            i := 1;
            WHILE Table(i,1) /= -1 LOOP
                IF Table(i,1) = 1 AND Table(i,2) = 1 THEN
                    SELECT
                        L1S1.len_msg(Table(i,3),self);
                        Table(i,1) := -1;
                    OR
                        DELAY XXXXX;
                        NULL;
                    END SELECT;
                ELSIF Table(i,1) = 1 AND Table(i,2) = 2 THEN
                    SELECT
                        L1S1.ack_msg;
                        Table(i,1) := -1;
                    OR
                        DELAY XXXXX;
                        NULL;
                    END SELECT;
                ELSIF Table(i,1) = 1 AND (Table(i,2) = 3 OR
Table(i,2) = 4 ) THEN
                    SELECT
                        IF L2S1 ' CALLABLE THEN
                            L2S1.over(Table(i,2),Table(i,3));
                        END IF;
                        Table(i,1) := -1;
                    OR
                        DELAY XXXXX;
                        NULL;
                    END SELECT;
                ELSIF Table(i,1) = 1 AND Table(i,2) = 5 THEN
                    SELECT
                        L1S1.stop;
                        Table(i,1) := -1;
                    OR
                        DELAY XXXXX;
                        NULL;
                    END SELECT;
                ELSIF Table(i,2) = 1 THEN
```

```
                    SELECT
                        L1S(i).len_msg(Table(i,3),self);
                        Table(i,1) := -1;
                    OR
                        DELAY XXXXX;
                        NULL;
                    END SELECT;
                ELSIF Table(i,2) = 2 THEN
                    SELECT
                        L1S(i).ack_msg;
                        Table(i,1) := -1;
                    OR
                        DELAY XXXXX;
                        NULL;
                    END SELECT;
                ELSIF Table(i,2) = 3  OR Table(i,2) = 4 THEN
                    SELECT
                        IF L2S(i)' CALLABLE THEN
                            L2S(i).over(Table(i,2),Table(i,3));
                        END IF;
                        Table(i,1) := -1;
                    OR
                        DELAY XXXXX;
                        NULL;
                    END SELECT;
                ELSIF Table(i,2) = 5 THEN
                    SELECT
                        L1S(i).stop;
                        Table(i,1) := -1;
                    OR
                        DELAY XXXXX;
                        NULL;
                    END SELECT;
                END IF;
                i := i + 1;
            END LOOP;
            compact;
        OR
            TERMINATE;
        END SELECT;
END CommunicationProcess;

--------------------------------------------------------------------------------

TASK TYPE Second1 IS
```

```
    ENTRY len_msg(s1,Pi : IN INTEGER);
    ENTRY ack_msg;
    ENTRY stop;
    ENTRYstart(ary1:item3;ary2:item4;pcount : INTEGER);
END;
TASK BODY Second1 IS
    d : INTEGER := INTEGER'LAST;    -- current shortest path
    num : INTEGER := 0;  -- unacknowledged messages
    pred : INTEGER;          -- most current predecessor
    self : INTEGER;
    ack : INTEGER := 0;
    s : INTEGER := 0;
    w : item4;
    count : INTEGER;
    XXXXX : INTEGER;  -- time to delay
    successor : item3;
    done : BOOLEAN := false;

BEGIN
    ACCEPT start(ary1 : item3;ary2 : item4;pcount : INTEGER) DO
        successor := ary1;
        count := pcount;
        w := ary2;
    END;
    CP(1).idself(1);
    FOR i IN 2..N LOOP
        IF successor(i) THEN
            CP(1).msg(i,1,w(i));
            num := num + 1;
        END IF;
    END LOOP;
    CP(1).msg(-1,-1,-1);
    LOOP
        SELECT
            ACCEPT len_msg(s1,Pi : INTEGER) DO
                pred := Pi;
                IF s1 < 0 THEN
                    done := true;
                    s := s1;
                ELSE
                    ack := 1;
                END IF;
            END;
            IF ack = 1 THEN
```

```
                CP(1).msg(pred,2,0);
                CP(1).msg(-1,-1,-1);
                ack := 0;
            END IF;
        OR
            ACCEPT ack_msg DO
                num := num - 1;
                IF num = 0 THEN done := true; END IF;
            END;
        OR
            ACCEPT stop DO
                count := count - 1;
            END;
        OR
            WHEN done =>
                FOR i IN 2..N LOOP
                    IF successor(i) THEN
                        CP(1).msg(i,5,0);
                        successor(i) := false;
                    END IF;
                END LOOP;
                CP(1).msg(-1,-1,-1);
                IF count = 0 THEN EXIT; END IF;
        END SELECT;
    END LOOP;
    Tester.message0(s,d,pred,num);
END Second1;
```

```
TASK TYPE Second IS
    ENTRY len_msg(s,Pi : IN INTEGER);
    ENTRY ack_msg;
    ENTRY stop;
    ENTRY start(ary1:item3;ary2:item4;id : INTEGER;pcount : INTEGER);
END;
TASK BODY Second IS
    d : INTEGER := INTEGER'LAST;    -- current shortest path
    num : INTEGER := 0;  -- unacknowledged messages
    pred : INTEGER;          -- most current predecessor
    self : INTEGER;
    ack : INTEGER;
    count : INTEGER;
    w : item4;
    successor : item3;
```

```
    done : BOOLEAN := false;
    saves , savepi : INTEGER;
BEGIN
    ACCEPT start(ary1 : item3;ary2 : item4;id , pcount : INTEGER) DO
        count := pcount;
        successor := ary1;
        w := ary2;
        self := id;
    END;
    CP(self).idself(self);
    LOOP
        SELECT
            ACCEPT len_msg(s,Pi : IN INTEGER) DO
                ack := 0;
                saves := s;
                savepi := Pi;
            END;
            IF saves < d THEN
                IF num > 0 THEN
                    CP(self).msg(pred,2,0);
                    CP(self).msg(-1,-1,-1);
                END IF;
                pred := savepi;
                d := saves;
                FOR i IN 1..N LOOP
                    IF successor(i) THEN
                        CP(self).msg(i,1,d+w(i));
                        num := num + 1;
                    END IF;
                END LOOP:
                IF num = 0 THEN
                    CP(self).msg(pred,2,0);
                END IF;
            ELSIF saves >= d THEN
                CP(self).msg(pred,2,0);
            END IF;
            CP(self).msg(-1,-1,-1);
        OR
            ACCEPT ack_msg DO
                num := num - 1;
            END;
            IF num = 0 THEN
                CP(self).msg(pred,2,0); CP(self).msg(-1,-1,-1);
            END IF;
```

```
            OR
                ACCEPT stop DO
                    count := count -1;
                    done := true;
                END;
            OR
                WHEN done =>
                    FOR i IN 2..N LOOP
                        IF successor(i) THEN
                            CP(self).msg(i,5,0);
                            successor(i) := false;
                        END IF;
                    END LOOP;
                    CP(self).msg(-1,-1,-1);
                    IF count = 0 THEN EXIT; END IF;
            END SELECT;
        END LOOP;
        Tester.message1(d,pred,num,self);
    END Second;

-------------------------------------------------------------------------------
  L1S1 : Second1;
  L1S : ARRAY (2..N) OF Second;
  CP : ARRAY (1..N) OF CommunicationProcess;
  BEGIN
      LOOP
          EXIT WHEN status /= NR;
      END LOOP;
  END Layer1Second;

-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
  PROCEDURE Layer2Second(Tester : IN OUT GlobalTester) IS

-------------------------------------------------------------------------------
      TASK CommunicationProcess IS
          ENTRY msg(to1,mtype1,w1 : IN INTEGER);
          ENTRY idself(id : INTEGER);
      END;
      TASK BODY CommunicationProcess IS
          self : INTEGER;
          tctr : INTEGER;
          to : INTEGER;
```

```
    i : INTEGER;
    Table : ARRAY(1..N,1..3) OF INTEGER;
-------------------------------------------------------------------------------
    PROCEDURE compact IS
        i : INTEGER;
        j : INTEGER;
        swap : ARRAY(1..N,1..3) OF INTEGER;
    BEGIN
        j := 1;
        FOR i IN 1..N LOOP
            IF Table(i,1) /= -1 THEN
                swap(j,1) := Table(i,1);
                swap(j,2) := Table(i,2);
                swap(j,3) := Table(i,3);
                j := j + 1;
            END IF;
        END LOOP;
        swap(j,1) := -1;
        i := 1; j := 1;
        WHILE swap(j,1) /= -1 LOOP
            Table(i,1) := swap(j,1);
            Table(i,2) := swap(j,2);
            Table(i,3) := swap(j,3);
            i := i + 1; j := j + 1;
        END LOOP;
        Table(i,1) := -1;
        tctr := i;
    END compact;
-------------------------------------------------------------------------------
BEGIN
    -- initialize Table to -1 and self
    LOOP FOR i IN 1..N LOOP
        Table(i,1) := -1; Table(i,2) := -1;
        Table(i,3) := -1;
    END LOOP;
    ACCEPT idself(id : INTEGER) DO
        self := id;
    END;
    tctr := 1;
    LOOP
        SELECT
                ACCEPT msg(to1,mtype1,w1: IN INTEGER) DO
                    to := to1;
                    Table(tctr,1) := to1;
```

```
                    Table(tctr,2) := mtype1;
                    Table(tctr,3) := w1;
                    tctr := tctr + 1;
                END;
                LOOP
                    EXIT WHEN to = -1;
                    ACCEPT msg(to1,mtype1,w1: IN INTEGER) DO
                        to := to1;
                        Table(tctr,1) := to1;
                        Table(tctr,2) := mtype1;
                        Table(tctr,3) := w1;
                        tctr := tctr + 1;
                    END;
                END LOOP;
            OR
                i := 1;
                WHILE Table(i,1) /= -1 LOOP
                    IF Table(i,1) = 1 AND Table(i,2) = 1 THEN
                        SELECT
                            L1S1.len_msg(Table(i,3),self);
                            Table(i,1) := -1;
                        OR
                            DELAY XXXXX;
                            NULL;
                        END SELECT;
                    ELSIF Table(i,1) = 1 AND Table(i,2) = 2 THEN
                        SELECT
                            L1S1.ack_msg;
                            Table(i,1) := -1;
                        OR
                            DELAY XXXXX;
                            NULL;
                        END SELECT;
                    ELSIF Table(i,1) = 1 AND ((Table(i,2) = 3) OR
(Table(i,2) = 4 )) THEN
                        SELECT
                            IF L2S1 ' CALLABLE THEN
                                L2S1.over(Table(i,2),Table(i,3));
                            END IF;
                            Table(i,1) := -1;
                        OR
                            DELAY XXXXX;
                            NULL;
                        END SELECT;
```

```
ELSIF Table(i,1) = 1 AND Table(i,2) = 5 THEN
    SELECT
        L1S1.stop;
        Table(i,1) := -1;
    OR
        DELAY XXXXX;
        NULL;
    END SELECT;
ELSIF Table(i,2) = 1 THEN
    SELECT
        L1S(i).len_msg(Table(i,3),self);
        Table(i,1) := -1;
    OR
        DELAY XXXXX;
        NULL;
    END SELECT;
ELSIF Table(i,2) = 2 THEN
    SELECT
        L1S(i).ack_msg;
        Table(i,1) := -1;
    OR
        DELAY XXXXX;
        NULL;
    END SELECT;
ELSIF (Table(i,2) = 3)  OR (Table(i,2) = 4) THEN
    SELECT
        IF L2S(i)'CALLABLE THEN
            L2S(i).over(Table(i,2),Table(i,3));
        END IF;
        Table(i,1) := -1;
    OR
        DELAY XXXXX;
        NULL;
    END SELECT;
ELSIF Table(i,2) = 5 THEN
    SELECT
        L1S(i).stop;
        Table(i,1) := -1;
    OR
        DELAY XXXXX;
        NULL;
    END SELECT;
END IF;
i := i + 1;
```

```
                END LOOP;
                compact;
        OR
                TERMINATE;
        END SELECT;
END CommunicationProcess;
```

```
TASK TYPE SecondII_1 IS
    ENTRY over(mtype , id: IN INTEGER);
    ENTRY start(s1,num1 : IN INTEGER;ary1, predecessor: item3);
END;
TASK BODY SecondII_1 IS
    msg : INTEGER;    -- message to be sent
    num : INTEGER;    -- unacknowledged messages
    s : INTEGER;      -- distance received
    self : INTEGER;
    backup : item3;
    predary : item3;
    change : BOOLEAN := true;
    successor : item3;
    ------------------------------------------------------------------------
    FUNCTION moretodo(ary : item3)  RETURN BOOLEAN IS
    done : BOOLEAN := false;
    i : integer := 1;
    BEGIN
        WHILE NOT done AND i <= N LOOP
            IF ary(i) THEN
                done := true;
            END IF;
            i := i + 1;
        END LOOP;
        RETURN done;
    END moretodo;
    ------------------------------------------------------------------------
BEGIN
    ACCEPT start(s1,num1 : IN INTEGER;ary1,predecessor: item3) DO
        s := s1;
        num := num1;
        predary := predecessor;
        successor := ary1;
        backup := successor;
    END;
```

```
            self := 1;
            CP(1).idself(1);
            IF s < 0 THEN
                msg := 3;
             ELSE
                msg := 4;
            END IF;
            LOOP
                SELECT
                    IF change THEN
                        FOR i IN 2..N LOOP
                            -- for all successors send over message.
                            -- But wait only xxxxx seconds for
                             -- a rendezvous.
                             IF successor(i) THEN
                                    CP(1).msg(i,msg,1);
                                    successor(i) := FALSE;
                             END IF;
                        END LOOP;
                        CP(1).msg(-1,-1,-1);
                        IF NOT moretodo(successor) THEN
                            change := false;
                        END IF;
                    END IF;
                OR
                    WHEN over'COUNT > 0 =>
                        ACCEPT over(mtype ,id: IN INTEGER) DO
                            IF msg = 4 AND mtype = 3 THEN
                                msg := 3; d := INTEGER'FIRST;
                                change := true;
                                successor := backup;
                            END IF;
                            predary(id) := false;
                        END;
                OR
                    IF NOT moretodo(successor) AND NOT moretodo(predary)
THEN
                        EXIT;
                    END IF;
                    END SELECT;
            END LOOP;
            Tester.message2(s,msg,self);
        END SecondII_1;
```

```
---------------------------------------------------------------------------
        TASK TYPE SecondII IS
            ENTRY over(mtype ,id: IN INTEGER);
            ENTRY start(d1,num1 : IN INTEGER;id : INTEGER;ary1,predecessor
: item3);
        END;
        TASK BODY SecondII IS
            num : INTEGER;  -- unacknowledged messages
            d : INTEGER;    -- current shortest path
            msg : INTEGER;      -- message received and sent later
            successor : item3;
            backup : item3;
            predary : item3;
            change : BOOLEAN := true;
            self : INTEGER;
            ---------------------------------------------------------------
            FUNCTION moretodo(ary : item3) RETURN BOOLEAN IS
            done : BOOLEAN := false;
            i : integer := 1;
            BEGIN
                WHILE NOT done AND i <= N LOOP
                    IF ary(i) THEN
                        done := true;
                    END IF;
                    i := i + 1;
                END LOOP;
                RETURN done;
            END moretodo;
            ---------------------------------------------------------------
        BEGIN
            ACCEPT start(d1,num1 : IN INTEGER;id : INTEGER;ary1
,predecesor: item3) DO
                d := d1;
                num := num1;
                self := id;
                predary := predecessor;
                successor := ary1;
            END;
            CP(self).idself(self);
            -- accept over message from predecessor
            ACCEPT over(mtype , id: IN INTEGER) DO
                predary(id) := false;
                msg := mtype;
                IF num > 0 AND msg /= 3 THEN
```

```
                msg := 3;
            END IF;
            IF msg = 3 THEN d := INTEGER'FIRST; END IF;
        END;
        LOOP
            SELECT
                -- if some task is waiting to rendezvous then accept its attempt
                WHEN over'COUNT > 0 =>
                        ACCEPT over(mtype , id: IN INTEGER) DO
                        IF msg = 4 AND mtype = 1 THEN
                            msg := 3; d := INTEGER'FIRST;
                            change := true;
                        END IF;
                        predary(id) := false;
                    END;
            OR
                IF change THEN
                    FOR i IN 2..N LOOP
                        -- for all successors, send over message.
                        IF successor(i) THEN
                                CP(self).msg(i,msg,self);
                                successor(i) := FALSE;
                            END IF;
                    END LOOP;
                    CP(self).msg(-1,-1,-1);
                    IF NOT moretodo(successor) THEN
                        change := false;
                    END IF;
                END IF;
            OR
                IF NOT moretodo(successor) AND NOT moretodo(predary)
THEN EXIT END IF;
            END SELECT;
        END LOOP;
        Tester.message2(d,msg,self);
    END SecondII;

-------------------------------------------------------------------------------
  L2S1 : Second1;
  L2S : ARRAY (2..N) OF Second;
  CP : ARRAY (1..N) OF CommunicationProcess;
  BEGIN
      LOOP
          EXIT WHEN status /= NR;
```

```
        END LOOP;
    END Layer2Second;
    ------------------------------------------------------------------------
    ------------------------------------------------------------------------
Tester : GlobalTester;
Status : statustype;
BEGIN
    status := NR;
    Layer1Primary(Tester);
    IF status = GE THEN
        status := NR;
        Layer1Second(Tester);
    END IF;
    IF status = OK THEN
        status := NR;
        Layer2Primary(Tester);
        IF status = GE THEN
            status := NR;
            Layer2Second(Tester);
        END IF;
    END IF;
    IF status = GE THEN
        -- Error
    END IF;
END DSPA;
```

BIBLIOGRAPHY

ANDE81 Anderson, T. and P.A. Lee, "Fault-Tolerance, Principles and Practice," Prentice-Hall International, Englewood Cliffs NJ, 1981.

BARN84 Barnes, J. G. P.,"Programming in Ada," Addison-Wesley Publishers Ltd., London, U.K., 1984.

BENA82 Ben-Ari, M. , "Principles of Concurrent Programming," Prentice-Hall International, Englewood Cliffs NJ, 1982.

CHAN82 Chandy, K.M. and J. Misra, "Distributed Computations on Graphs: Shortest Path Algorithms," Comm. of ACM,Nov. 1982,vol.25,No.11, pp. 833-837.

ELRA83 Elrad, T. and N. Francez, "Decomposition of Distributed Programs into Communication-Closed Layers," The Science of Computer Programming, No. 2, 1983, pp. 155-173.

ELRA84 Elrad, T.,"A Practical Software Development for Dynamic Testing of Distributed Programs," IEEE Proceedings of the International Conf. on Parallel Processing, Bellaire, MI, Aug. 1984, pp. 388-392.

FRAN80 Francez, Nissim, "Distributed Termination,"ACM TOPLAS, Jan.1980, Vol. 2, No. 1,pp. 42-55.

GERM84 German, Steven M.,"Monitoring for Deadlock and Blocking in Ada Tasking,"IEEE Trans. on Soft. Engin.,Nov. 1984,Vol. SE-10, No.6, pp. 764-777.

GERT86 Gerth, R. and L. Shrira,"Proving Noninteraction : An Optimized Approach," submitted to ICACP, 1986.

HOAR78 Hoare, C. A. R.,"Communicating Sequential Processes,"CACM, August 1978, Vol. 21, No. 8, pp.666-677.

LEE87 Lee, P. and C. Malik, "Distributed Shortest Path Algorithm: Constraints and Efficiency Issues in CSP and Ada," To appear ACM South Central Regional Conference, Lafayette, La. , November 19-21, 1987.

LEE88 Lee, Pen-Nan,"Violation Detection and Recovery of Distributed Programs' Safety Properties," To appear 7th Annual IEEE Phoenix Conf. on Computers and Communications, March 16-18, 1988.

MOIT83 Moitra, A.,"Synthesis of Communicating Processes," Proceedings of the Second Annual ACM Symp. on Principles of Dist. Comp., Montreal, Canada, Aug. 1983, pp. 123-130.

RAND75 Randell, Brian , "System Structure for Software Fault Tolerance," IEEE Trans. on Software Engin., June 1975, Vol. SE-1, No. 2, pp.220-232.

USDD81 U.S. Department of Defense, "Programming Language Ada: Reference Manual," Vol. 106, Lecture Notes in Computer Science, Springer-Verlag, New York, 1981.