SEL-76-043

THE OPTIMAL PLACEMENT OF DYNAMIC RECOVERY CHECKPOINTS IN RECOVERABLE COMPUTER SYSTEMS

by

Wayne Alan Warren-Angelucci

December 1976

Technical Report #126

This research was supported by the Defense Advanced Research Projects Agency under ARPA Order No. 2494, Contract No. MDA903-76C-0093, by the National Science Foundation under research grant MC573-07973-A1, 2, and by Joint Services Electronics Program: U.S. Army, U.S. Navy, and U.S. Air Force under Contract N-000 14-67-A-O 112-060 1.

DIGITAL SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES

STANFORD UNIVERSITY • STANFORD, CALIFORNIA

SEL-76-043

THE OPTIMAL PLACEMENT OF DYNAMIC RECOVERY CHECKPOINTS IN RECOVERABLE COMPUTER SYSTEMS

by

Wayne Alan Warren-Angelucci

December 1976

Technical Report #126

DIGITAL SYSTEMS LABORATORY

Department of Electrical Engineering
Stanford University
Stanford, California

This research was supported by the Defense Advanced Research Projects Agency under ARPA Order No. 2494, Contract No. MDA903-76C-0093, by the National Science Foundation under research grant MC573-07973-A1,2, and by Joint Services Electronics Program: U.S. Army, U.S. Navy, and U.S. Air Force under Contract N-00014-67-A-0112-0601.

ABSTRACT

Reliability is an important concern of any computer system. No matter how carefully designed and constructed, computer systems fail. The rapid and systematic restoration of service after an error or malfunction is always a major design and operational goal. In order to overcome the effects of a failure, recovery must be performed to go from the failed state to an operational state. This thesis describes a recovery method which guarantees that a computer system, its associated data bases and communication transactions will be restored to an operational and consistent state within a given time and cost bound after the occurrence of a system failure.

This thesis considers the optimization of a specific software strategy - the rollback and recovery strategy, within the framework of a graph model of program flow which encompasses communication interfaces and data base transactions. Algorithms are developed which optimize the placement of dynamic recovery checkpoints. Presented is a method for statically pre-computing a set of optimal decision parameters for the associated program model, and a run-time technique for dynamically determining the optimal placement of program recovery checkpoints.

ACKNOWLEDGEMENTS

Many people have been helpful during my years at Stanford and especially during the research and preparation of this thesis.

My advisor, Professor Vinton Cerf, deserves a great deal of thanks for his constant support and encouragement. He always had helpful suggestions when the going was slow and showed a great deal of patience with the numerous false starts and mistakes. I appreciate his suggestion of the general area of research that is investigated in this thesis.

I am grateful to Carol Hankins and C. Peter McCullough for their aid and assistance in producing the hardcopy of this manuscript.

I owe thanks to several organizations for their support during my studies and research at Stanford: the Joint Services Electronics Program: U.S. Army, U.S. Navy, and U.S. Air Force under Contract N-00014067-A-0112-0601, the Defense Advanced Research Projects Agency under ARPA Order No. 2494, Contract No. MDA903-76C-0093, and the National Science Foundation under research grant MC573-07973-A1,2.

Finally, I want to thank my wonderful wife, Danielle. She has been a boundless source of encouragement for me. Her enthusiasm was highest when mine was lowest, and she was always patient with my occasional preoccupied periods. For all of this I

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Chapter 1
INTRODUCTION

1.1 Reliability and Recoverability

The concept of reliable computing has existed for a long time, but it has remained almost exclusively the preserve of the hardware designer. Hardware structures have been developed which can continue to provide the required facilities despite occasional failures, either transient or permanent outages of internal components and modules [1,2 3.

Since the term *reliable system* can have many different meanings, it is important to clearly establish just what is desired. One does not try to build a completely non-failing device. Instead, one introduces redundancy into systems of intrinsically unreliable components. This redundancy may be in hardware - additional hardware modules, or it may be in *time* - additional software, or a combination of both. Given this redundancy, the attempt is to build a system which will recover automatically within a given time period. Recovery is defined as the continuation of system functions, after the incidence of an error, with data integrity. In a total system environment, it is a problem requiring both hardware and software aids. The fault must be diagnosed and if a broken hardware module is at fault, then it needs to be removed from the system. For both transient and solid failures, data must be reconstructed to a consistent state before restarting.

In many applications it is not necessary to operate continuously and perfectly. The needed reliability of a computer system is a function of the task which is being performed. A computer failure while running a numerical analysis program is annoying at worst. However, in such areas as real-time process control, spacecraft guidance, and air traffic control a computer failure can be catastrophic. When human lives are at stake it is imperative that systems perform reliably. For these situations it is sufficient to operate correctly most of the time, so long as outages are infrequent, fixed with minimal human intervention, and most importantly that the system recover within a maximum time limit.

How one copes with infrequent brief outages depends on what one is trying to do. For tasks which are tightly coupled to real time requirements, such as a real time process control application, a method is to choose checkpoints at which to record the state of the system, so that one can always recover by restarting from the checkpoint just preceeding an outage [3]. Other applications with tighter real-time constraints may only tolerate outages of several seconds, or milliseconds before the system suffers catastrophic, unrecoverable failure. Thus an aircraft guidance system might at times, tolerate only the briefest downtime, whereas an airline reservation system could adapt to downtimes of a considerably longer duration, before the network's general operation would be jeopardized.

Occasionally, despite all efforts, a system will break so catastrophically that it will be unable to recover. Given that there is sufficient redundancy in a system, a goal is to reduce the probability of such a system failure to the probability of failure of all redundant components. The presence of operable system components, however, is not sufficient to guarantee that operation will be resumed. In addition, the software must be able to survive the transients accompanying the failure, re-configure, adapt to the remaining hardware, restore all faulty data to a consistent state, and continue processing.

In certain applications one is also concerned with maintaining privacy and security along with reliability. Security is concerned with protecting a system from an active external agent who seeks to defeat system objectives. Reliability means the ability of a system to overcome or recover from random errors. Security and reliability are related, and often the techniques for providing reliability will interfere with the maintenance of a secure system. These interdependencies will be elaborated later when the system model is described.

The major approach to computer reliability proposes a redundant system design and studies the interaction of the various redundancy techniques. The redundancy may exist in the form of extra modules, such as triple modulo redundancy [TMR] techniques [4,5,6,7], redundant software, such as Randell's method of acceptance tests with alternate recovery blocks [8], or the use of *extra* time to perform the function of maintaining system integrity.

The technology of reliable computing encompasses theory and techniques of fault detection and correction, modelling, analysis, synthesis, and the architecture of fault-tolerant systems and their evaluation [9]. Reliability and recoverability cannot be added on to a computer system. An iterative process must be used in design. The final implementation of the recovery process will be the result of evaluation of the best possible data integrity assurance at the minimum cost in both hardware and software.

In a typical recoverable computer system, there are four major tasks to perform. The first is fault or error detection. The second is the identification of the fault. in hardware, a data transaction error, or possibly identifying the fault as transient and non-recurring. The third is the modification of the system to eliminate the cause of the fault, and fourth is the system restart after reconfiguration.

Hardware checking and diagnostics can be used for assumed failure modes, but they must be supplemented. In addition, program errors must be discovered. This can be done by audit programs interleaved with operational programs. If software information is to be audited, it must be redundant and be able to satisfy consistency relations.

One use of audit programs is to check hardware by its proper execution of operational programs. A second use is to check for data integrity, for example, the consistency of data base information. A third is to check the validity of the information necessary for the supervisory program: the queues, tasks, system directories, buffers and other system resources allocated by software.

Recovery programs are the software equivalent of hardware retry. Similarly, audit algorithms are analogous to hardware checking. The better these algorithms are, the greater the information integrity and the more valid is the recovery information. Also, without accurate diagnosis of the cause of the fault, system reconfiguration will be inaccurate and much potential fault tolerance will be wasted. Audit routines are extremely useful, but are environment dependent. They perform hardware checking of the components and data transaction consistency. They are principally responsible for information integrity of the system. Recovery routines are activated by the audit routines. They reconfigure the system, and then restore it to a consistent state via checkpoint and rollback techniques.

1.2 Hardware Reliability

Reliability enhancement is achieved through better components and by adding functional redundancy to the hardware modules. There are two types of functional redundancy: fault masking redundancy and standby redundancy. Masking redundancy is achieved by implementing a function or module so that it is inherently error correcting, for example TMR techniques. With standby redundancy, spare modules are switched into the system when working modules break down. The process of applying tests and determining whether the computer is fault free or not is generally known as fault detection or checkout. Fault location and isolation is the process of identifying the failures within the smallest possible set of components.

In order to avoid complete systems failures, a failed component must be repaired or replaced before its backup also breaks. The system must therefore report all failures. It must be possible to remove and replace any component while the system continues to run. The system should absorb repaired or newly introduced modules gracefully.

1.2.1 Reliable Hardware Systems - TMR

Triple Modulo Redundancy [4,7] was one of the earliest methods suggested for obtaining a reliable system from less reliable components. The system output of Fig. 1.1 is the majority of three identical components.

If only one of the components is in error, the system output will not be in error, since the majority of components will not be in error. Thus, the system can tolerate errors in any one component. These errors may be transient or permanent.

1.3 Hybrid Reliability

Software recovery after fault detection with hardware self-repair is a hybrid utilization of reliability techniques. Various strategies are used to reduce the impact of interruptions or malfunctions both to the system and to the user. Operating System 360 as used in Model 65 [3] is equipped with a set of programs called the recovery management support which embodies a number of methods. The recovery methods depend on the nature of the malfunction. In the input/output area, rereading of input data with parity errors is common. If errors persist even after repeated retries, the system could consider reconstruction of damaged data (error correction) if possible. In the case of processor error, the instruction may be retried if feasible (if the operands were not modified by the instruction). The most important technique OS 360/65 provides is checkpoints in all programs so that programs can be rolled back to a previous state and computation resumed.

1.3.1 Hybrid Systems - JPL STAR

The JPL STAR (Self Testing And Repair) computer system, as seen in Fig. 1.2

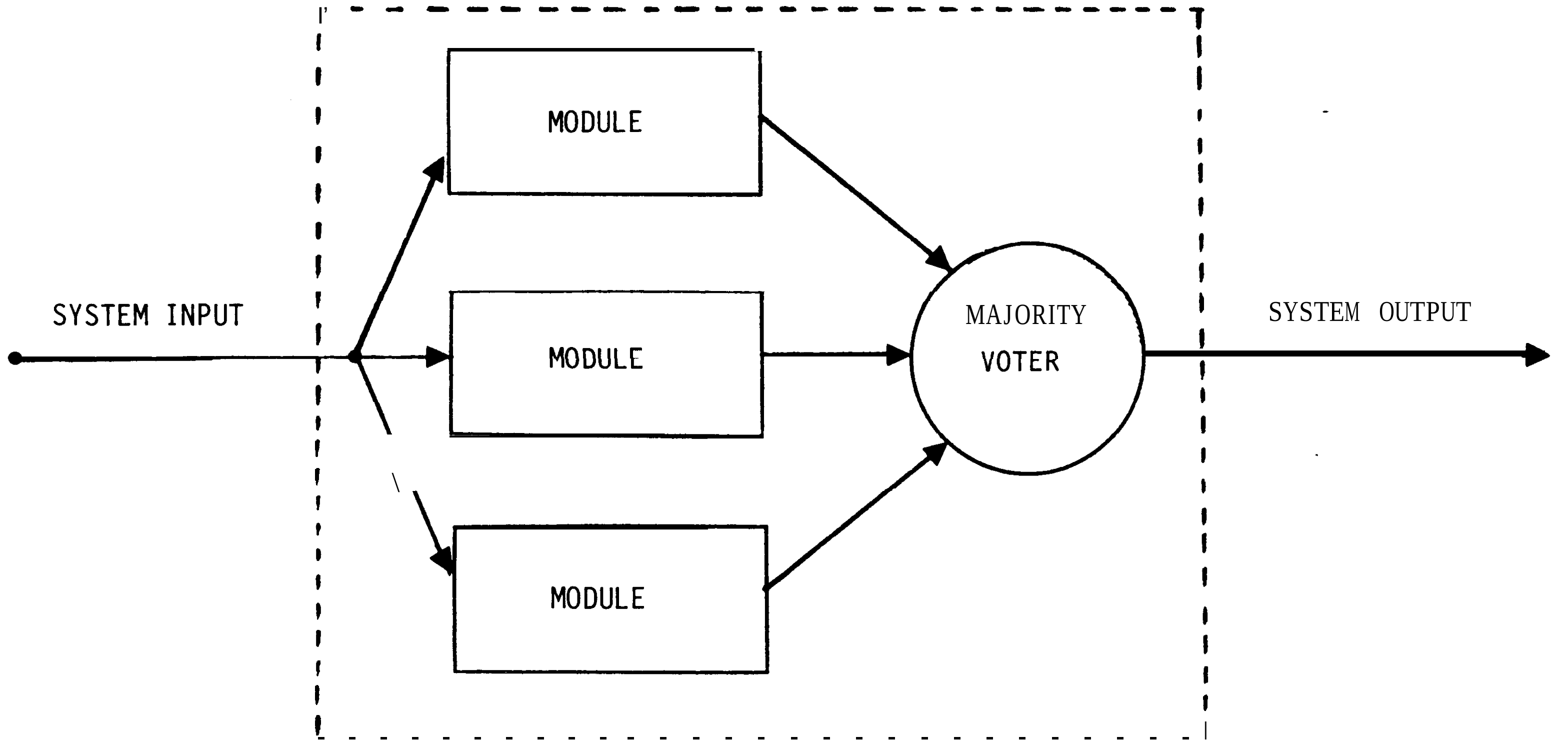


Figure 1.1 - TMR BLOCK DIAGRAM

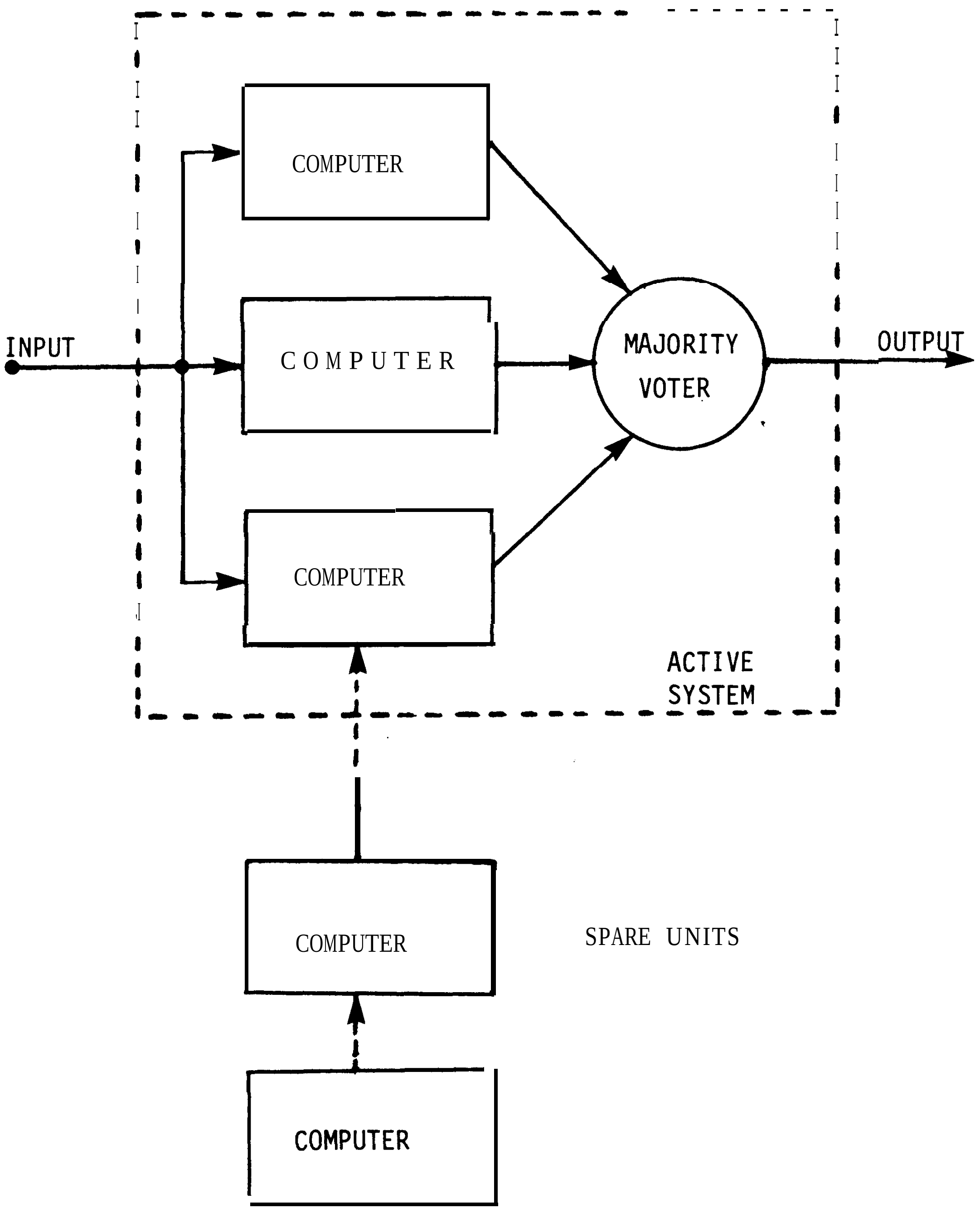


Figure 1.2 - JPL STAR HYBRID COMPUTER BLOCK DIAGRAM

obtains reliability by using TMR and spares [4].

The n spares are inactive and not powered on. If at any point in processing, one of the three active modules disagrees with the majority, the disagreeing module in the minority is switched out and replaced by a spare. The spare must be powered up and loaded. One method of loading is to use rollback and load the component with the last saved error free state, and resume computation from that point [10]. If at most, one component (module) fails during a rollback cycle, and if the vote taker is error free, the system is fail safe until all the spares are used up.

1.3.2 Hybrid Systems - PLURIBUS

BBN has constructed a multiprocessor computer system, Fig. 1.3, which achieves both increased operating power and gains increased system reliability through parallelism and redundancy [1,11]. Their system architecture is known as Pluribus. The system consists of processor units, memory units and input/output units. Each unit is in itself a communication buss providing a physical housing, power and cooling, and a communication discipline provided by a buss arbiter. All processor busses are coupled to all memory busses and all input/output busses, likewise all memory busses are coupled to all input/output busses.

The Pluribus system provides extra copies of every vital hardware resource, and *isolation* between copies so that any single component failure will impair only one copy, leaving a potentially runnable machine. It also provides software facilities necessary to survive transients stemming from failure and to adapt to running on the new hardware configuration.

1.4 Software Reliability

Methods have been developed to improve reliability primarily by means of software

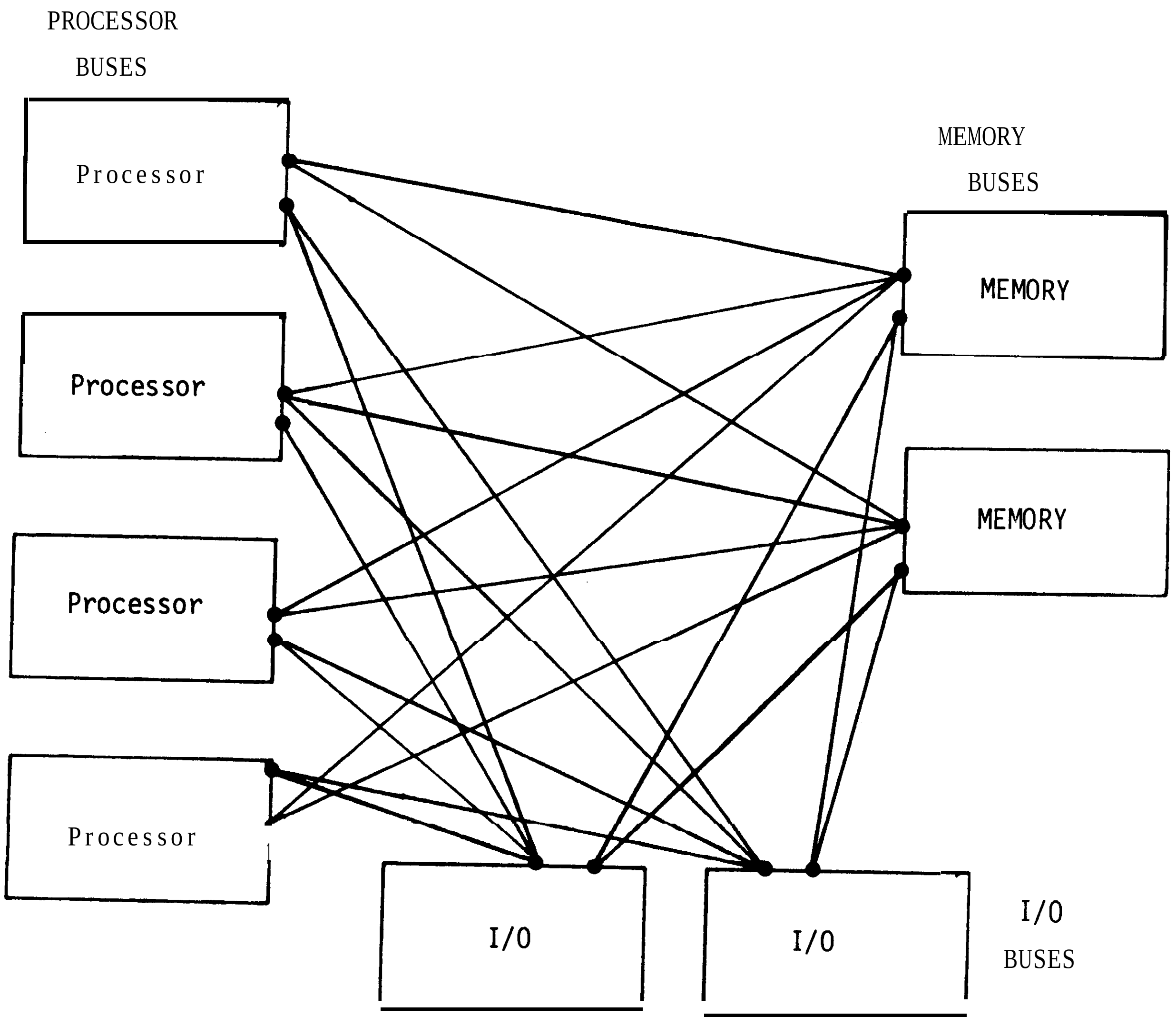


Figure 1.3 - BBN PLURIBUS BLOCK DIAGRAM

[8,12,13]. The cost associated with software methods is generally the additional time and storage required for processing.

In many real time systems it is necessary to recover rapidly from an error. One way of achieving quick recovery is to fix the cause of the error (assuming that it was not transient), and then rollback and restart the program at a previously saved error free state. If an error is detected while a program is being processed and if the error cannot be corrected immediately, it may be necessary to run the entire program again. The time lost in running the program may be substantial and in some real time applications, critical. Software methods for enhancing reliability assume that the systems programs are written correctly. Software techniques utilizing incorrect programs will not improve system reliability.

Since software is an expensive item, and software errors have become very costly, a need arises to validate and verify the correct operation of a software package before it is committed to regular use. Several approaches to the formal validation of programs have been studied [16,17,18]. These methods either put considerable burden of the validation on the programmer or require that the input/output assertions provided in the program be verified by a sophisticated theorem proving mechanism. An abstract model of computations in a program and a method of proving that a specific program will always run properly is provided by King [19]. The complexity of most of the program verifying techniques indicates the need for much simpler methods which could provide *partial* validation of large programs.

1.4.1 <u>Reliable Software Systems - Randell</u>

Randell [8] has developed a method for structuring programs by the use of recovery *blocks*. This is illustrated in Fig. 1.4. His aim is to provide the dependable error detection and recovery facilities which can cope with errors caused by software design inadequacies, particularly in the system software, rather than the malfunctioning of hardware components.

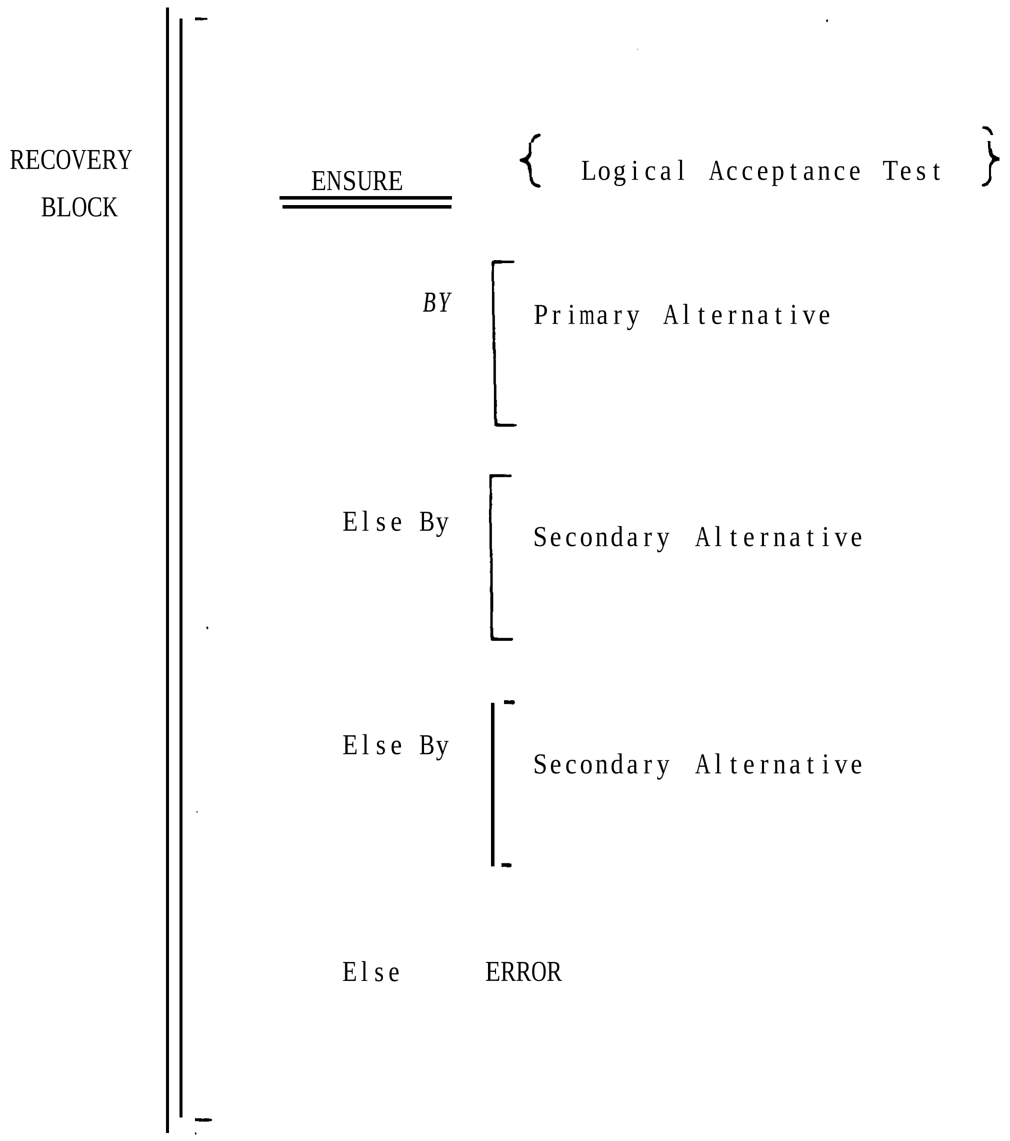


Figure 1.4 = RANDELL'S RECOVERY BLOCK SCHEME

Randell's scheme is software analogous to hardware *standby sparing.* As the system operates, tests are made on the acceptability of the results generated by each software module. Should one of these checks fail, a spare software module is switched in to take the place of the erroneous module. The spare software module is not a copy of the main software component, but one utilizing an alternate and independent design, so that it, hopefully, can cope with the circumstances which caused the main component to fail. The technique uses recovery *blocks,* in which acceptance tests of task functions are ensured by primary or else by a number of secondary alternatives.

His recovery block scheme incorporates a solution to the problem of switching to the use of the spare component and of repairing damage done to non-local data via a recursive cache structure. If the backed up program has modified global variables, these variables must be restored to their previous values, or the program could operate on incorrect data when the it is restarted. If the variables that a program has modified were used by another program, then the program that used the modified variables must also be backed up.

Randell's model, though, does limit concurrent processing, all data transactions with a data base, and communications with other external processes.

1.4.2 Software Reliability - Russell's Extensions

Russell [12] has extended Randell's work on recovery blocks and recursive cache by presenting a system design that supports the restoration of system state in a system of asynchronous communicating parallel processes. He provides send/receive primitives which implement interprocess communications through messagelists, illustrated in Fig. 1.5. Also provided are recovery primitives which perform the consistent state restoration of the system.

The complexity of state restoration is analyzed, and is shown to be dependent upon

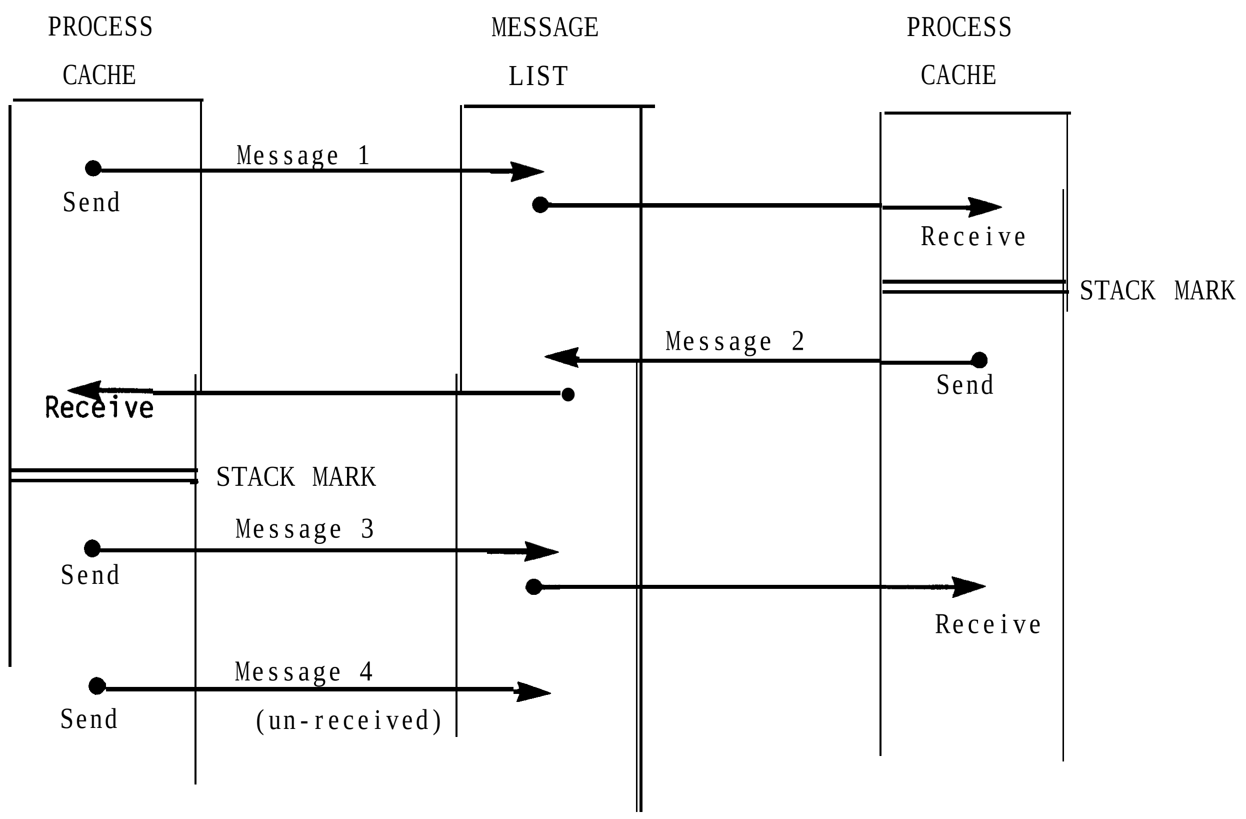


Figure 1.5 - RUSSELL'S PROCESS CACHE AND MESSAGELIST SCHEME

the structure of the messagelist and on its interconnection with the processes. More efficient recovery is possible if the system is constrained to insert *stackmarks* into the process cache before executing the send/receive operations. Russell finds bounds for the amount of state restoration which must be performed to restore the system to a previously consistent recovery point after the occurrence of an error.

1.4.3 Software Reliability - Chandy's Work

Chandy [13,141 has presented a model of computation for process control systems. Each job run by the system is partitioned into several tasks, Fig. 1.6, by the programmer. The programmer must construct a task graph which represents the program control flow. Associated with each vertex of the graph is the maximum execution time for the corresponding task.

His graph models the synchronous program execution in which multiple edges leading away from a node represent possible control flow points of which only one may be taken at run time. Chandy's model arrives at a means of optimally inserting checkpoints along edges of program flow. His model includes neither concurrent processing nor transactions with an external data base or communications with external processes. It also requires a priori knowledge of node execution times.

1.5 Hardware - Software Tradeoffs

Consider an aerospace system such as an air-traffic control system. The system has a specific goal which must be accomplished within a certain specified amount of time. A large penalty is incurred if the system does not accomplish its mission. A lateness penalty is incurred if the time taken to accomplish the goal exceeds this limit. The longer the time taken the larger the penalty.

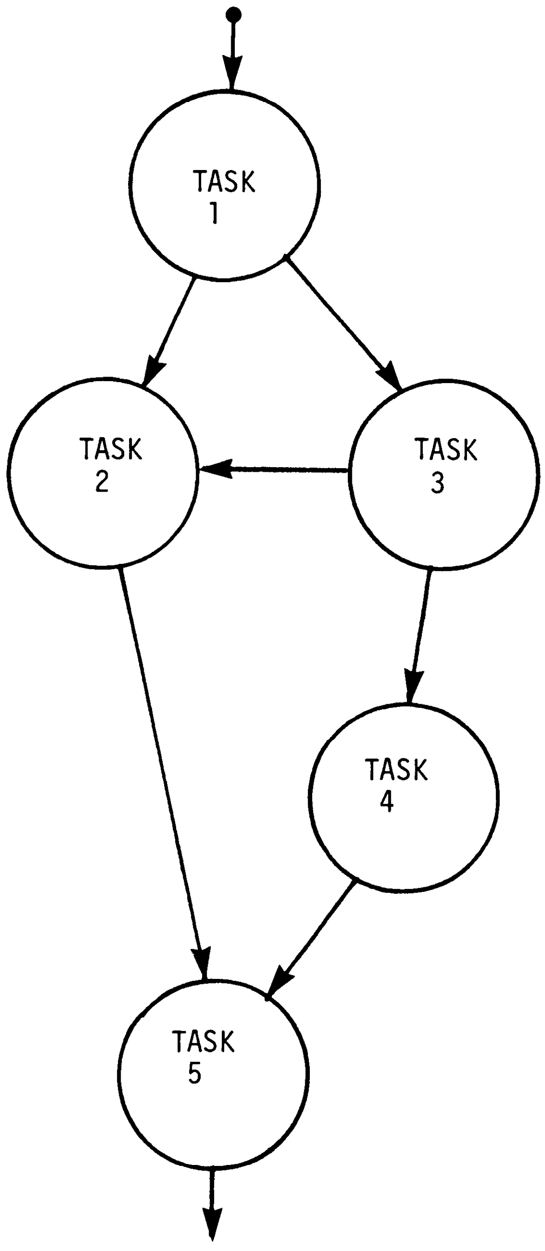


Figure 1.6 - CHANDY'S GRAPH MODEL OF PROGRAM BEHAVIOR

Several models have been constructed for designing reliable machines from intrinsically less reliable components by using redundancy. These are hardware methods. The cost of a hardware method is the cost required to build and maintain the redundant hardware. The cost associated with software methods of achieving reliability is generally the additional time and space required for processing, and possibly the actual manpower cost associated with providing this software.

In some systems, ‘software methods have to be ruled out since the amount of time available to complete a task is too short to permit methods which require additional time. In other cases, the longer the system takes, the more expensive are the consequences.

Studies have been done by Ramamoorthy, Chandy and Cowan [15] which attempt to construct a framework in which hardware and software methods can be compared for cost effectiveness. In essence, their method compares the costs of delays introduced by time redundancy techniques with the costs of hardware in hardware redundancy schemes. They analyze TMR, hybrid, and TMR with standby spares (self-purging), obtaining techniques for computing a set of indices for comparison of reliability methods. However, the problem of selecting the optimal mix of redundancy strategies for a system is very difficult because of the numerous cost-effectiveness parameters which can be adjusted.

1.6 Summary

A computer system for error recovery must provide four capabilities:

1. a means for detecting that an error occurred,

2. a means for locating and diagnosing the error,

3. a means for correcting any adverse affects the error has caused, and

4. a means for reconfiguring the system so that the error does not reoccur.
Retrying a failed operation will succeed if the error was a transient error.

In the remainder of this thesis, detection, location, diagnosis, and reconfiguration are not considered. The optimal restoration of the correct system state is studied.

The algorithms and programs used for the recovery scheme are assumed to be error free and to function properly.

1.7 Organization of Thesis

In Chapter 2, "A Model of Program Structure for Rollback and Recovery," the particular characteristics of a highly available computer system are presented, along with a model of program behavior which includes communication interfaces and data base transactions. The concept of the statically pre-computed decision parameter set which minimizes the expected program execution time is introduced, and an algorithm for its use at run-time is presented.

In Chapter 3, "Algorithm Which Optimizes the Insertion of Recovery Checkpoints," the MERT algorithm is presented and proven to minimize the total expected run time of the program.

Chapter 4, "Examples Illustrating the Dynamic Insertion of Recovery Checkpoints," presents a (typical) program graph which is analyzed by the MERT algorithm. This analysis determines the optimal decision parameter set, which is then used to illustrate the run-time program behavior for several different runs of this same program.

Chapter 2
A MODEL OF PROGRAM STRUCTURE FOR ROLLBACK AND RECOVERY

Program checkpointing and rollback is a method of enhancing computer system reliability. Program checkpointing is the process of making a copy of program state in secondary storage. Rollback is the re-loading of this state upon the occurrence of an error, and the restart of the system.

The objective and constraints may vary considerably from system to system. The system being considered is assumed to have high availability. However, if an error does occur, it must recover very rapidly since a delay in performing system functions may have catastrophic results. The objective is to assure that every recovery be rapid. For our system, an explicit constraint is assumed: the interval of time taken to recover must not exceed a given quantity, M time units. In actual practice M will depend on the system, may vary from job to job in a given system, and may depend upon the actual stage in processing for a particular job. Our system is assumed to have sufficient processing power to perform its primary task and to support the overhead which is associated with rollback and recovery. The objective of our analysis is to minimize this associated rollback and recovery overhead, with the constraint that recovery never exceed M time units.

2.1 Optimal Placement of Rollback Points

The checkpointing strategy may be static or dynamic. Static checkpointing requires carrying out checkpointing at fixed intervals regardless of their immediate necessity. In a dynamic checkpointing environment, the placement of checkpoints will vary from one run of a program to the next. This variation will depend upon the dynamic runtime characteristics of the program. Dynamic checkpointing yields higher system availability than static checkpointing because it takes into account the actual rollback and recovery requirements.

The optimal placement of recovery checkpoints necessitates that the programmer analyze the program flow and make estimates on certain branching parameters, task execution times, etc. The more often a program is run, the more cost-effective and beneficial is the optimal placement of recovery checkpoints. Programs whose total processing time is shorter than a maximal crucial recovery time do not need recovery checkpoints. A program which is worth analyzing for the optimal placement of rollback points will have a combination of these attributes:

* It must be crucial to the application of the program that error recovery be accomplished quickly and systematically.

* It is necessary to maintain correct operation, imperative that errors be detected and corrected.

* The same program must be run a number of times, a *production* program.

* The program will require a substantial amount of processing time.

There are many application areas which possess these attributes:

* Real time process control of expensive or dangerous components.

* Applications where human lives are endangered by extended system downtime, such as air traffic control.

* Applications in which many people are dependent upon the continuous service of an essential or expensive commodity, such as electronic funds transfer (EFT) in banking or an airline reservation system.

* Spacecraft guidance, navigation, and life support systems.

There are several different parameters to consider when deciding the *optimal* placement of recovery checkpoints. The choice and placement to insert a checkpoint depends upon the importance of speedy error recovery. In certain real time applications it is crucial that a program reliably run to completion bounded by a fixed maximal time limit. In other applications, the loss incurred due to a system failure is only the computer time wasted.

2.2 Program Graph Model

2.2.1 Program Graph

Our algorithm for the optimal placement of recovery checkpoints uses a sequential graph model to describe a program. Similar graph models have been used for the analysis of program structure and behavior [20,21,22,23]. We require that a programmer analyze his program and represent it as a sequence of *tasks*. This analysis could be done manually from a flow chart, or could be accomplished automatically with the aid of an analysis program. A task will consist of a number of machine instructions, and will involve an amount of processing time which is bounded by the maximal recovery time M.

Let a program be represented by a directed graph, as in Fig. 2.1, where each *node i* in the graph corresponds to a *task i* in the program, and *edge (ij)* exists if task j may directly succeed task i with probability p_{ij}.

2.2.2 Program Running Time

The analysis makes use of estimates made by the programmer on the expected amount of processing time, t_i, required by a task i. It is impossible to design an algorithm, which, given any program, determines the time that may be required to process each task in the program. However, it is possible for a programmer to obtain estimates of average or worst case bounds for the tasks of his particular program. These times could be obtained through a measurement system. In many computer installations, programmers submit estimates of the maximum time required to process their jobs. It is important to note that in installations where a

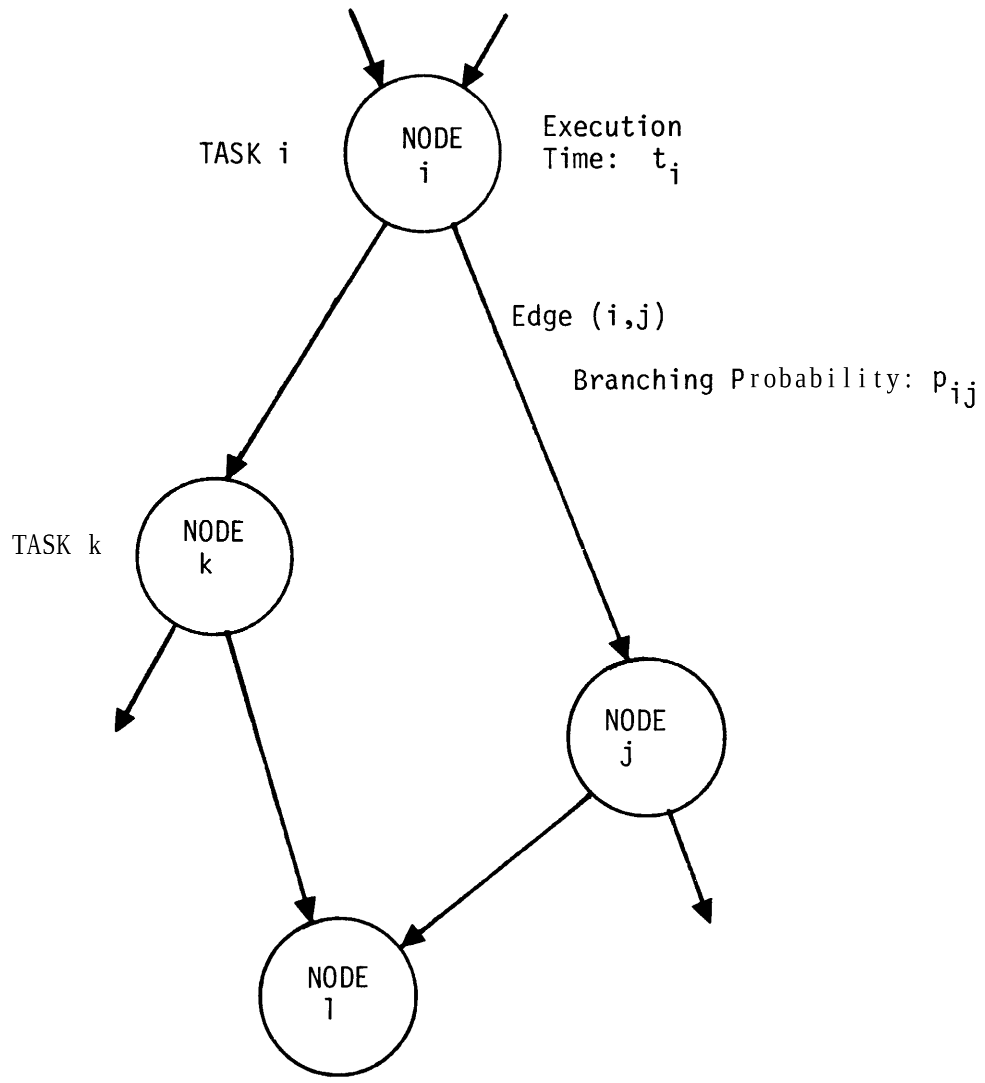


Figure 2.1 - GRAPH MODEL OF PROGRAM BEHAVIOR

programmer is allowed to specify recovery checkpoints, he must make estimates of this sort, and then make intuitive decisions based on these estimates. Our objective is to clarify and formalize this decision making process. The accuracy of the decisions clearly depends on the accuracy of the estimates. The decision to insert recovery checkpoints depends on the importance of speedy error recovery: the penalty incurred if a program does not run to completion in a prescribed amount of time.

Obtaining a program graph from a program is not inexpensive. The program must be analyzed and estimates made of several parameters. In many cases, the advantage gained in having tailor made recovery checkpoints is not worth the time spent to obtain a program graph. In these cases static checkpoints at fixed intervals are sufficient. However, in those cases where the costs of slow efror recovery are- high, the advantage of dynamic recovery checkpoints outweighs the time spent to construct the program graph. We are concerned with cases (Section 2.1) of this latter type.

2.2.3 Program State

At any stage in the processing of a program, certain information is required by the program for computation to proceed successfully. A *state* at any stage in the processing of a program will be defined as the information (program variables, state of the input/output devices, secondary storage) which may be subsequently used by the program.

At each edge (i,j), one may dynamically choose to insert a *recovery checkpoint*. If a checkpoint is inserted on edge (i,j), then after task i is completed and before task j is begun, the state of the system is saved in secondary storage. Any state saved prior to this (i,j) rollback point is accumulated, allowing subsequent recovery attempts to make use of multiple recovery checkpoints (Section 2.4). Thus, when an error occurs, an attempt is made to restart the program from the most recently saved state.

2.2.4 Branching Probabilities

Associated with each program branch is the probability, p_{ij}, that branch (i,j) will be followed. For each node i, it follows that

$$\sum p_{ij} = 1$$

Furthermore, the probabilities p_{ij}, are assumed to be fixed and independent of the way the program reached the particular branch (i,j). This will be recognized as a Markov model assumption. Although this is not always valid, it is a simplifying assumption to aid the analysis. If we do not assume a Markov model, the resulting analysis is overly complex [23].

2.2.5 Acyclic Program Graph

The sequencing of tasks may change from one run to the next due to the conditional branching probabilities p_{ij}. In the graph model, it is assumed that no task is repeated. The program graph G is an acyclic directed graph. If there is a loop in the program, then each iteration may be modeled as a distinct task, or the iterations may be combined into a sequence of one or more tasks. Beizer [23] presents a method for transforming a cyclic program graph into one which is acyclic.

2.2.6 Error Latency and Detection

An error which is not detected as soon as it occurs may propagate. For example, if an erroneous input is used to update a data element, the updated item will also be in error. When an error is detected, it is not generally possible to ascertain when the error occurred, nor the amount of error propogation. The error latency is the period of time between the occurrence of an error and its detection. The distribution of error latency depends upon the method used for error detection. If error detection occurs intermittently at a fixed interval T, then the error latency is not likely to exceed T. The error latency distribution influences the amount of error propagation. If an error has a short lifetime, it is less likely to be used to update other data items, and therefore less likely to propagate.

Error detection may be performed continuously or intermittently. Parity checking is one example of continuous error detection since it can proceed as long as the system is available. O&her techniques may perform integrity/consistency checking only at discrete intervals. The ability to *localize* the extent of error has a beneficial impact on the recovery process.

Suppose an error occurs while a task is being processed, and suppose it is not diagnosed (Fig 2.5). If a checkpoint occurs immediately after the task is complete, then the information that is saved at the checkpoint will be erroneous. Subsequently, if an error is diagnosed, this erroneous information will be loaded, and the system will continue processing from the recovery checkpoint. Eventually the same error will be diagnosed again. If the same error is detected after rolling back, we can conclude that an undiagnosed error occurred before the last recovery checkpoint, or that there exists a permanent malfunction which the system reconfiguration has not corrected. If an undiagnosed error has occurred, we have the potential of rolling back again, to a previously saved recovery checkpoint, in an attempt to reload a correct copy of the program state.

2.3 Data Set Interactions

The total state of the system which must be saved at rollback recovery checkpoints consists of both program state (memory, registers, etc.) and the complete state of all peripheral devices which interact and exchange information with the program. Let us refer to all interacting external devices as ***data sets.***

2.3.1 Data Sets

Data sets include all input and output devices, such as user terminals, measuring transducers, system data bases, etc. If a system failure occurs, and operation returns to a previously established rollback recovery point, it must be possible to insure that all data sets are returned to consistency with the program state at this rollback point.

Rolling back data sets implies that:

1. input devices must be able to furnish again the identical sequence of data,
2. output results, some of which may be erroneous, must be revoked,
3. additions, deletions and updates to data bases must be undone, and
4. dialogue with users at remote terminals must be duplicable.

2.3.2 Transaction Journals

In order to provide for this backup capability, the system must continuously maintain a record of pertinent transactions which occur with each data set. In its simplest form, the transaction journal is a record of input and output transactions performed by the program. Each item is written onto secondary storage before it is processed. For an update action, a typical journal entry might consist of the item name which is being updated, its old value, and its new value. The content and use of these journals is dependent upon the characteristics of the individual data set.

Three types of transaction journal need to be generated by the system so that data set rollback may take place:

2.3.2.1 Backup Journals

The *backup journal* is used to restore the data set to the earlier state which existed at the last rollback point. Backup journals provide a record of those input and output transactions between the program and data set which modify the state of the data set. The backup journal need only record those transactions which result in an update being made to the data set. Individual entries in the backup journal include that information necessary to undo state modifications, such as additions and deletions to a data base.

2.3.2.2 Revoke Journals

The *revoke journal* provides the ability to either revoke any erroneous output which was issued by the real data set since the last recovery checkpoint, or to indicate the extent of this erroneous data so that an (external) agent might take appropriate actions to recall or undo the effects of this output data. Entries in the revoke journal would indicate that output which might be potentially erroneous due to the occurrence of a system error.

2.3.2.3 Replay Journals

The *replay journal* provides the ability to simulate the replay of a non-repeatable input data set, such as a reader, measuring transducer, etc. Entries into the replay journal would include all those input data transactions which the data set could not again furnish to the program.

2.3.3 Virtual Data Sets

To facilitate rollback and restart, it is desirable to treat all classes of data sets in a similar and general manner. For our model this is accomplished by providing an *interface process* between the real (physical) data set and the program. This interface process creates a *virtual data set* as seen by the program. A virtual data set is one which possesses the same attributes as the real data set, but in addition, is capable of being backed up to a previously defined state upon the detection of a system error. This capability is independent of the actual physical attributes and limitations of the real data set. This is shown in Fig. 2.2.

2.3.3.1 Virtual Data Set Interface Processes

The *virtual data* set capability is provided by the data set *interface process,* whose structure and actions are tailored to the characteristics of the real data set. The interface process and its attendant transaction journal supplement the real data set, creating those capabilities necessary for the correct execution simulating a virtual data set. Both the interface process and the transaction journal are tailored specifically to create a virtual data set interface for the associated real data set.

Depending upon the characteristics of the real data set, the interface process makes use of the transaction journal to provide this virtual capability.

2.3.3.2 Recovery Commands to the Interface Process

In order to ensure data set consistency during the rollback and recovery procedure,

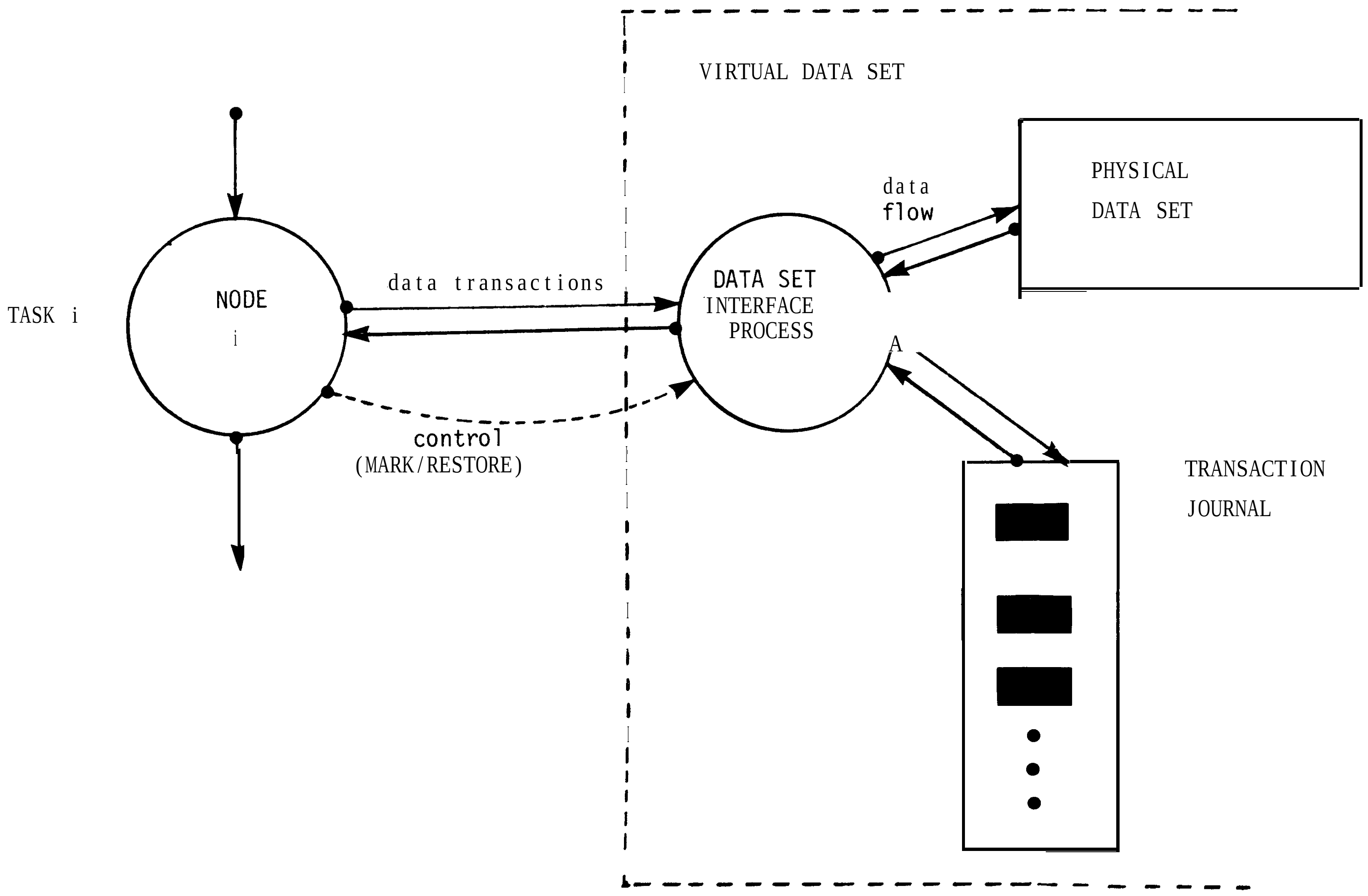


Figure 2.2 - Virtual Data Set Interface Process

all virtual data sets must recognize and act upon control commands from the program. These commands are:

1. MARK *(data set name)*
2. RESTORE *(data set name)*

When a recovery checkpoint is inserted in the program, say at edge (i,j), the program state is saved in secondary storage, and all virtual data sets are notified of this insertion by the MARK control command. Upon receipt of the MARK command, the virtual data set interface process takes the appropriate action necessary to ensure that the real data set, at a later point in time, may be restored to a state of consistency with this recovery checkpoint so that succeeding transactions may be again duplicated. Typical actions taken at the receipt of the MARK command might include clearing the previous journal entries, noting the current system clock time, etc. Succeeding interactions between the program and the virtual data set produce journal entries, the content of which is determined by the characteristics of the real data set.

Upon the occurrence of system error, the program state is reloaded from the latest rollback checkpoint, and the virtual data sets are notified of the error by the RESTART command. Upon reciept of the RESTART, the virtual data set takes the necessary action to restore its real data set, enabling subsequent input/output transactions with the program to be duplicated and replayed.

2.3.3.3 Classes of Data Sets

Let us divide the various types of data sets into equivalence classes, each of which possesses common characteristics as seen by the virtual data set interface process. The goal of this classification scheme is to systematically provide a virtual data set which:

1. exhibits the same operational characteristics as the real data set, during error-free system operation,

2. may be notified of the insertion of a program recovery checkpoint via a MARK command,

3. upon the receipt of a RESTORE command (at the occurrence of a system error) restores its real data set to the state consistent with the last rollback checkpoint, and

4. will furnish the identical sequence of input and output actions while the program is being rerun during the recovery process.

Data Sets Requiring No Transaction Journal

The simplest type of peripheral device to provide a virtual data set interface for is one which possesses internal state which is not modified by data transactions, and which is fully repeatable. This type of device will be read-only, and no rollback state restoration need be performed, nor transaction journals generated. A read only random access disk file, or a table lookup ROM are examples of this type of data set. The MARK and RESTORE commands are null operations for these devices.

Data Sets Requiring Backup Journals

Input/output transactions to a data set with internal state will be non-repeatable if the transaction modifies the internal state of the device. This state modification may be the result of an update or write action, such as updating a record in a random access disk file; or it may the result of physical movement, such as a read operation which repositions the input pointer of a magnetic tape unit.

A backup journal needs to be generated for this class of data set. The MARK command causes the current state of the device to be entered onto the journal so that subsequent state modifications may be undone. Each data transaction which updates this state must generate a journal entry which can later be used to undo the modification. Systems exist which make use of sophisticated backup mechanisms [24].

Data Sets Requiring Revoke Journals

This class of data set includes output devices which interact with and output information to an external destination, such as a lineprinter, display, etc. A revoke journal needs to be produced for this type of device so that upon the occurrence of a system error, that output which was issued since the last MARK command may be marked as erroneous. A revoke journal might also be used to take appropriate action to revoke those items which were issued in error.

The revoke journal entries must include all those output results which may be potentially erroneous upon the occurrence of a system error. For a particular device, if it is sufficient to note only the extent of the erroneous output which was issued since the last rollback recovery checkpoint, then a revoke journal does not need to be generated, and the interface process need only indicate which output will be re-issued.

Data Sets Requiring Replay Journals

Input devices which possess no internal state and which are not repeatable, require the generation of a replay journal. Such devices include keyboards, input samplers, card readers, and remote terminals. Individual entries into the replay journal include that information which is produced by the device and which is not repeatable. Upon the occurrence of a system error, the replay journal is used to furnish the same sequence of input data to the program.

If the data set is used interactively for both input and output, such as a remote user terminal, and the input received from the device is functionally dependent upon the output transactions issued to the device then the interface process should notify the terminal user, telling him to retransmit all messages sent after the message with such and such a serial number, which was the last correct transaction preceding the last rollback checkpoint. By using serial numbers, it could check carefully that it is not causing a file to be updated twice.

When the user's dialogue is interrupted, it is sometimes advantageous that he should be able to start it again where he left off. If it is a lengthy dialogue, for example the reordering of complex machinery, it is certainly desirable that he should not have to go back to the beginning. For this reason, checkpoints may be built into a dialogue structure. At intervals the decisions made up to that point in the dialogue will be reviewed, and possibly changed. When the terminal user agrees that it is correct, they will be recorded.

In most terminal conversations, there are certain stages at which the set of decisions recorded up to that point can be agreed upon as correct. This can be regarded as a *closure* in the decision-making process. Sometimes it is merely an arbitrary stage in data entry. Periodically, at a natural closure the user should be given a recap of what has been established up to that point and asked to check it. When the user has agreed that this is correct, the system will process these transactions.

2.3.4 Processing and Storage Overhead

Furnishing these virtual capabilities will incur an overhead cost due to additional processing and secondary storage requirements. These overhead costs are:

C_{create} - the additional processing time required by the virtual data set interface for the creation and recording of the transaction journals.

$C_{storage}$ - the additional secondary memory required for storage of the data set transaction journals.

C_{memory} - the additional system memory required for the storage of the virtual data set interface process code and internal buffering.

When a system error occurs (RESTORE), the additional processing costs are:

$_{backup}$ - the additional processing time required to backup the data set to the point of the previous MARK command, undoing all state modifications.

$C_{revo\ kc}$ - the time and processing cost of revoking the erroneous output issued.

C_{replay} - the time and processing cost necessary to replay those input transactions which are not repeatable.

For the subsequent calculation of recovery load time (Section 2.3.5), let us define a processing time cost C_i which is the sum of the backup, revoke and replay times which are incurred during the execution of task i. C_i is the total per recovery RESTORE time for data sets due to interactions with task i of the program:

$$C_i = C_{backup} + C_{revoke} + C_{replay}.$$

2.3.5 Save and Load Time

The quantity of program state (memory in use, etc.) which must be saved when a recovery checkpoint is inserted may vary widely from one point in the program to the next. Assume that a sufficient amount of secondary storage exists, so that it will always be possible to save the entire program state. The secondary storage used may be disk, drum, magnetic tape, or other, more advanced media [e.g. laser photostore]. The media used will, of course, affect the save time.

Associated with each edge (i,j) of the program graph, as seen in Fig. 2.3, are two numbers: S_{ij} and L_{ij}. If the execution of task j follows task i, then the state of the system after task i is completed and before task j is begun is the collection of the registers, program status and condition words, primary memory, and the state of all of the external data sets. The time taken to save the state at this point in the program secondary storage, create a recovery checkpoint, and reset the virtual data set interface process (via the MARK command) is S_{ij}. The *save time* S_{ij} may vary over other edges (i,j) of the program graph.

Given that a recovery checkpoint was established on edge (i,j) of the program graph, and that an error occurs during the execution of some succeeding task, say task k, the time taken to reload the program state from this established checkpoint is $reload_{ij}$.

The time taken for the data set interface to process this rollback (via the RESTORE command) is $T_{restore}$. This time, $T_{restore}$, includes any special action which the data set interface process must take for handling backing up, revoking and replay. $T_{restore}$ is the sum of the RESTORE time C_i, over all nodes i, which were processed since the last recovery checkpoint. For example, if a checkpoint were established at edge (a,b) of the program graph in Fig. 2.4., and recovery is to be performed during the execution of task f, then:

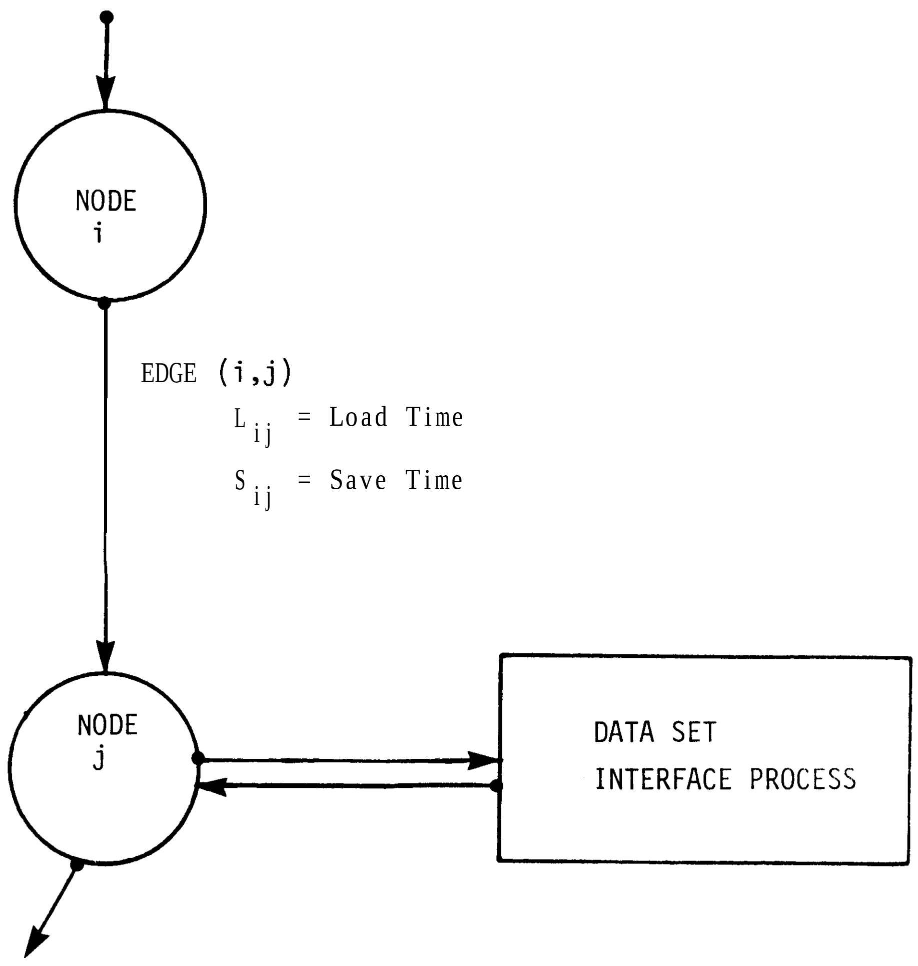


Figure 2.3 - PROGRAM SAVE AND LOAD TIME

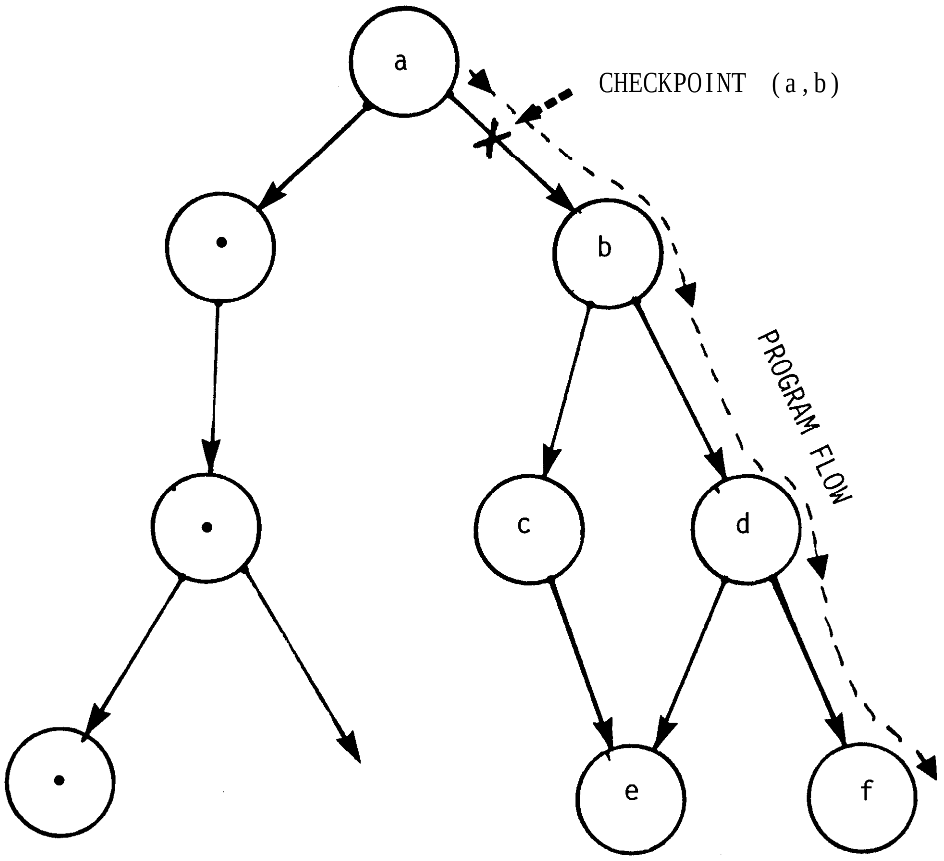


Figure 2.4 - SAMPLE CALCULATION OF RESTORE COST

$$T_{restore} = \sum_{i \in (b,d,f)} C_i$$

The total time taken to restore this particular system to consistency is the *load lime,* L_{df}:

$$L_{df} = reload_{df} + T_{restore}.$$

2.3 Recovery Time

Define the *recovery time* ***r*** at any point (in particular, at a point P, when the error is diagnosed) in the program to be the *interval* of time taken to:

1. reconfigure the system if the error was caused by a faulty hardware module,
2. restore the system to the consistent state of the most recent recovery checkpoint which contains the *correct* program state, and
3. rerun the program, from this state to point P.

Thus, if an error is detected at point P, the recovery time r is the *amount* of time *lost* due to the error. If the error was caused by a transient hardware fault, then the reconfiguration time is zero. If an error occurs while task i is being processed, and the error is detected before task i is completed, then the most recent recovery checkpoint will contain the correct system state. If this is not so, the system needs to be rolled back to a more previous checkpoint (Fig 2.5, and Section 2.2.6).

The recovery time r can be quickly determined during program execution. Define SysClock to be the value of the *system clock* at point P (when the error is diagnosed). If there are n previously saved recovery checkpoints on secondary storage, then the expected recovery time r at point P is:

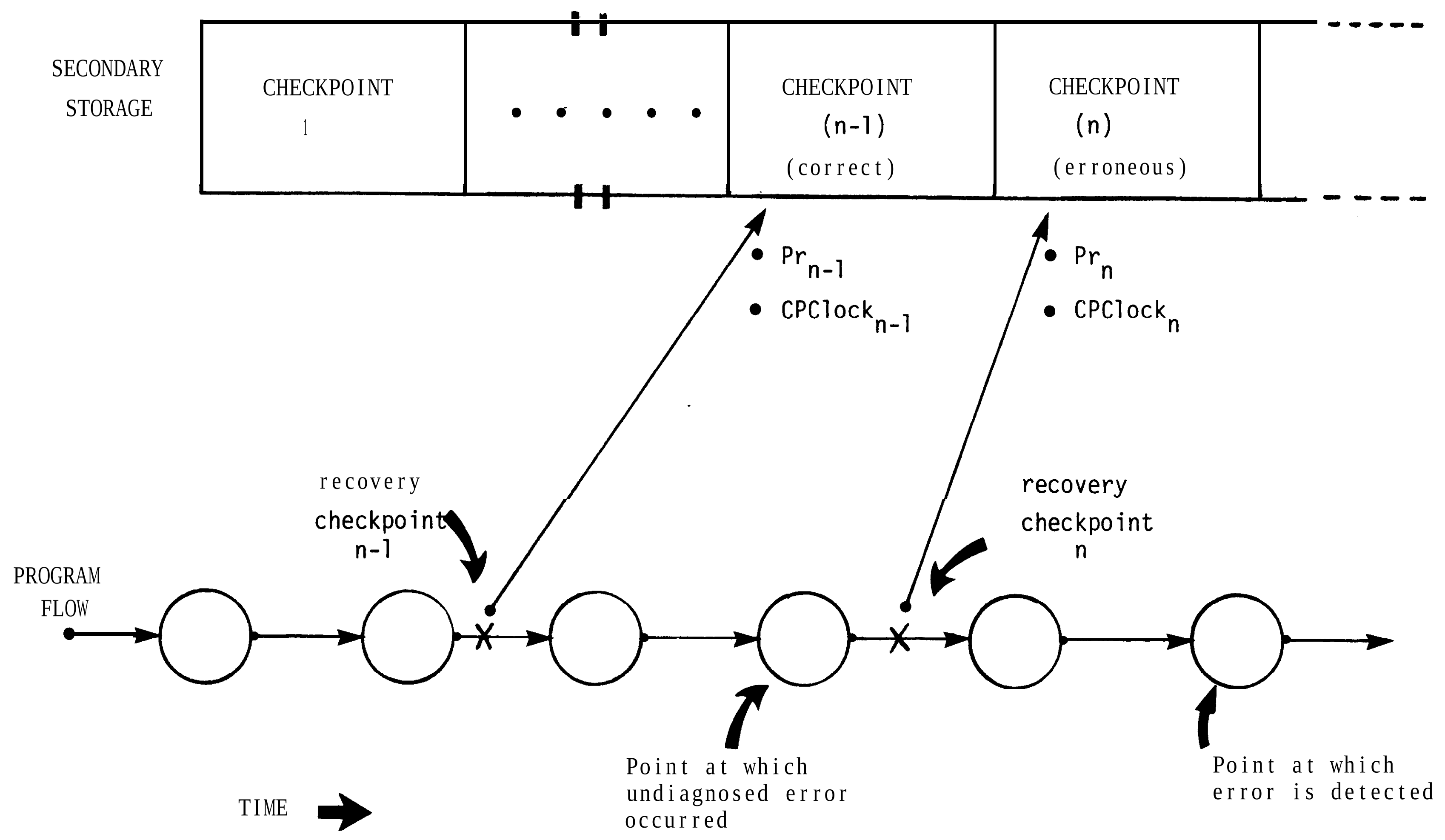


Figure 2.5 - OCCURRENCE OF UNDIAGNOSED ERROR

$$r = R + \sum_{i=1}^{n} \{SysClock - CPClock_i + Load_i\} * Pr_i$$

where:

R	- is the expected time to reconfigure the system hardware, perhaps amputating a faulty module. If the error was caused by a transient fault then $R = 0$.
$CPClock_i$	- is the system clock time of the i^{th} recovery checkpoint. If no recovery checkpoints were previously placed, then $CPClock_1$ is the value of the system clock at the very start of the program run.
$Load_i$	is the time taken to reload the program state from the i^{th} recovery checkpoint, and restore data sets to consistency with this rollback checkpoint (Section 2.3.5).
Pr_i	- is the probability of the i^{th} recovery checkpoint containing the correct state information.

If a reliable method of error detection (Section 2.2.6) is performed *before* program tasks complete, then it is possible to partition our system so that errors are not likely to propagate across task boundaries. If a reliable method of error/consistency detection is employed, then there will be a high probability that the most recently established recovery checkpoint will contain the correct state information, and $Pr_1 + Pr_2 + \ldots + Pr_{n-1} \approx 0$, and $Pr_n \approx 1$, so:

[illegible]

2.5 Optimal Decision Parameter - B_{ij}

It is not generally possible to predict the precise amount of time that a given program task will require. Thus, it is desirable to make the insertion of rollback points a dynamic procedure. On certain runs of a program, one may want a rollback point inserted on a particular edge (i,j), while on another run of the same program, with different data, it would not be optimal to place a rollback point on that same edge. The decision to insert rollback points should be quite simple so that it can be done in real time with little or no overhead.

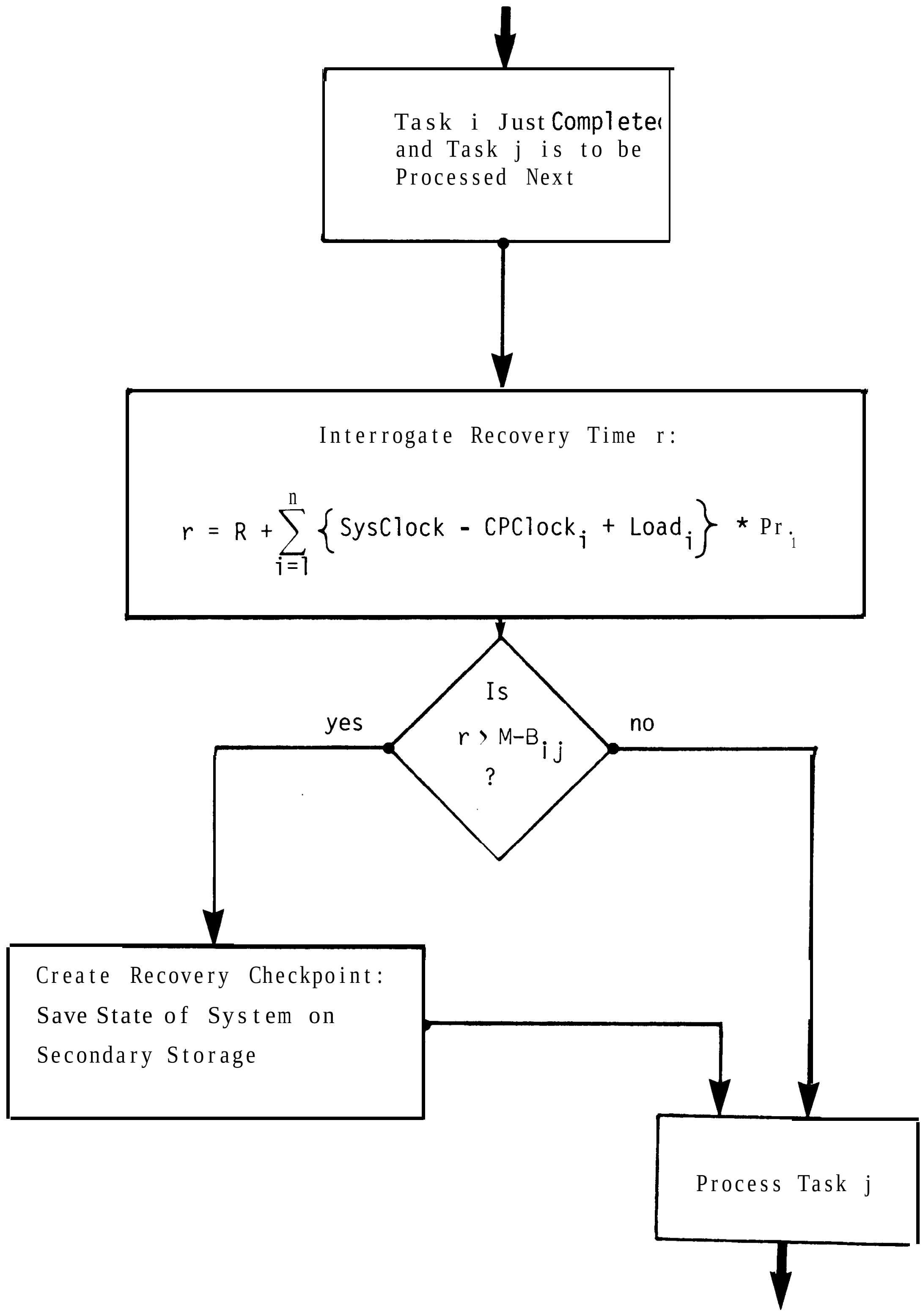
Suppose that at some point P in the program flow, that task i was just completed, and task j is to be executed next. Thus we lie on edge (i,j) of the program graph. Let **r** be the recovery time at this point P. Define the ***optimal decision parameter*** to be B_{ij}. Then, if $r > M - B_{ij}$ a recovery checkpoint should be inserted. If $r \leq M - B_{ij}$ choose not to insert a checkpoint. B_{ij} is a constant, and the set of all B_{ij} are computed ***before*** the program is run. Then, ***dynamically*** at runtime, after task i is completed and before task j is processed, the recovery time **r** is interrogated (Section 2.4) and a recovery checkpoint is inserted ***only*** if $r > M - B_{ij}$. **Figure** 2.6 illustrates this procedure. In general **r** will vary from one run of the program to the **next,** because the time taken to execute a particular task will depend on the input data to the program. Thus, the insertion of rollback points also varies from run to run, since the optimal decision is a function of r.

The analysis which follows in Chapter 3 determines the optimal placement of recovery checkpoints to:

1) minimize the total expected program running time, and

2) minimize the recovery overhead cost.

These algorithms statically compute the set of optimal decision parameters B_{ij}.



Task i Just Complete
and Task j is to be
Processed Next
Interrogate Recovery Time r:
$r = R + \sum_{i=1}^{n} \left\{ \text{SysClock} - \text{CPClock}_i + \text{Load}_i \right\} * \text{Pr}_i$
Is
$r > M-B_{ij}$
?
yes
no
Create Recovery Checkpoint:
Save State of System on
Secondary Storage
Process Task j

Figure 2.6 - RUNTIME INSERTION OF RECOVERY CHECKPOINTS

Chapter 3
ALGORITHM WHICH OPTIMIZES THE INSERTION OF RECOVERY CHECKPOINTS

3.1 Purpose of the MERT Algorithm - Minimization of the Expected Run Time

The purpose of this algorithm is to minimize the ***expected run time*** of the modeled system under the constraint that the ***expected recovery*** time must not exceed a bound of M time units. The ***interval*** M is assumed to be a constant, system defined bound. From our graph model of program flow, we will need to determine the task branching probabilities p_{ij}, and the expected execution time for task i, t_i.

3.2 Estimates of Program Behavior

3.2.1 Probability of Occurrence of Task Error

Other estimates of program behavior are needed for this analysis. Associated with each task is q_i, the probability that at least one error will occur during the processing of task i.

This probability, q_i, is primarily a function of the hardware which is supporting the execution of this task i. If the hardware support cannot be estimated, then q_i might be approximated by:

$$q_i = Q / n$$

where Q is the total probability of a system error occurring and n is the number of runnable tasks in the program.

If one can reasonably estimate the amount of time $t_{i,a}$ that task i is spending on hardware module a, and the probability of failure Q_a on module a, then a more reasonable estimate for q_i is:

$$q_i = \sum_{a \in H_i} (t_{ia} / t_i) * Q_a$$

where H_i is the set of all hardware modules supporting the execution of task i.

3.2.2 Expected Time Until Detection of a Task Error

Given that an error does occur during the interval in which task i is processing, let the random variable y_i be the expected time between the initiation of task i, and the detection of the error. The parameter y_i could be accurately estimated only by a substantial amount of program analysis and measurement. If a reliable method of error detection (Section 2.2.6) is performed *before* program tasks complete, then it is possible to partition our system so that errors are not likely to propogate across task boundaries. If this is so, y_i is bounded from above by the expected task execution time:

$$y_i \leq t_i$$

3.2.3 Summary of Program Behavior Estimates

Before proceeding with the minimization algorithm, let us briefly recap those estimates of program flow behavior which we will be using in the following analysis:

t_i	Expected time to execute task i correctly, given that no errors occur. A random variable (Section 2.2.2).
p_{ij} -	The probability (fixed, independent) that task j will follow task i - i.e., that edge (i,j) of the program graph will be taken (Section 2.2.4).
q_i	The probability of encountering at least one error during the execution of task i.
$1-q_i$ -	The probability of executing task i successfully.

y_i - The expected time between the initiation of task i and the detection of an error, if one occurs.

S_{ij} - The time taken to establish a recovery checkpoint on edge (i,j) (Section 2.3.5).

D_i - The expected recovery time if task i fails. This variable includes:

1. R, the system re-configuration time.

2. C_i, the sum of the backup, revoke and replay processing times which are incurred during the execution of task i (Section 2.3.4).

M - The maximum bound on the expected recovery time.

3.3 Definition: Expected Task Execution Time

Let us establish $E[t_i]$, the ***expected execution time for task i,*** given the probability of task error q_i.

Let $E[t_i]$ be characterized, by a geometric distribution, as:

$$E[t_i] = \sum_{k=0}^{00} (1-q_i)\, q_i^k \{t_i + \quad k(R + \quad y_i)\}$$

noting that:

$(1-q_i)\, q_i^k$ - is the probability of k failures followed by a successful execution of task i.

$t_i + k(R + y_i)$ - is the recovery time for k failures plus the expected time to successfully execute task i without errors.

Then:

$$E[t_i] = \sum_{k=0}^{\infty} (1-q_i)q_i^k \; t_i + \sum_{k=0}^{\infty} (1-q_i)q_i^k \; k(R + y_i)$$

$$= (1-q_i) \; t_i \sum_{k=0}^{\infty} q_i^k + (1-q_i) \; (R + y_i) \sum_{k=0}^{00} k \; q_i^k$$

$$= t_i + \{q_i \; (1-q_i) \; (R + y_i)\} \; d/dq \; (\sum q_i^k)$$

$$= t_i + \{q_i \; (1-q_i) \; (R + y_i)\} \; / \; (1-q_i)^2$$

so:

$$E[t_i] = t_i + \{ \; q_i \; (R + y_i) \; \} \; / \; (1-q_i)$$

3.4 MERT Algorithm

3.4.1 The Auxiliary f(r) and g(r) Functions

For each node i of the program graph **G**, let us define a function **$f_i(r)$**, which is the *minimum total* ***expected execution*** *time* of the program from the point *after* task i completes until the ***termination*** of the program. This minimum expected execution time includes:

1. the expected program task execution time, and

2. the time required to establish all required recovery checkpoints.

Note that f is a function of the expected recovery time r.

For each edge (i,j) of the program graph **G**, define a function $g_{ij}(r)$, which is also a function of the recovery time **r**. The $g_{ij}(r)$ function is the ***minimum expected total execution time*** of the program after task i completes until the termination of the program, ***if task j follows task i.*** Thus in the computation of $g_{ij}(r)$ it is implicit that the (i,j) program branch is being taken. The $f_i(r)$ and $g_{ij}(r)$ functions are illustrated in **Fig. 3.1.**

3.4.2 The MERT Algorithm

Define **fi(r)** = ∞ for all nodes i in the program graph, whenever r is greater than the ***maximum*** expected recovery interval, **M.**

Step 0

Define **fi(r)** = 0, when $r \leq M$, for all ***exit*** nodes i of the program graph G. A node with no successors is an ***exit*** node. After the function $f_i(r)$ has been determined for node i, consider node i to be ***labeled,*** else ***unlabeled.*** Thus, at this point, all ***exit*** nodes of the program graph are labeled.

kth Step: (k = 1,2,...)

If no unlabeled nodes remain on this kth step then terminate the algorithm, having computed the set of optimal decision parameters, $\{B_{ij}\}$. If there exists an *unlabeled* node i, which has all of its successor nodes ***labeled*** **(Fig 3.2),** then label node i by computing the function $f_i(r)$ as follows:

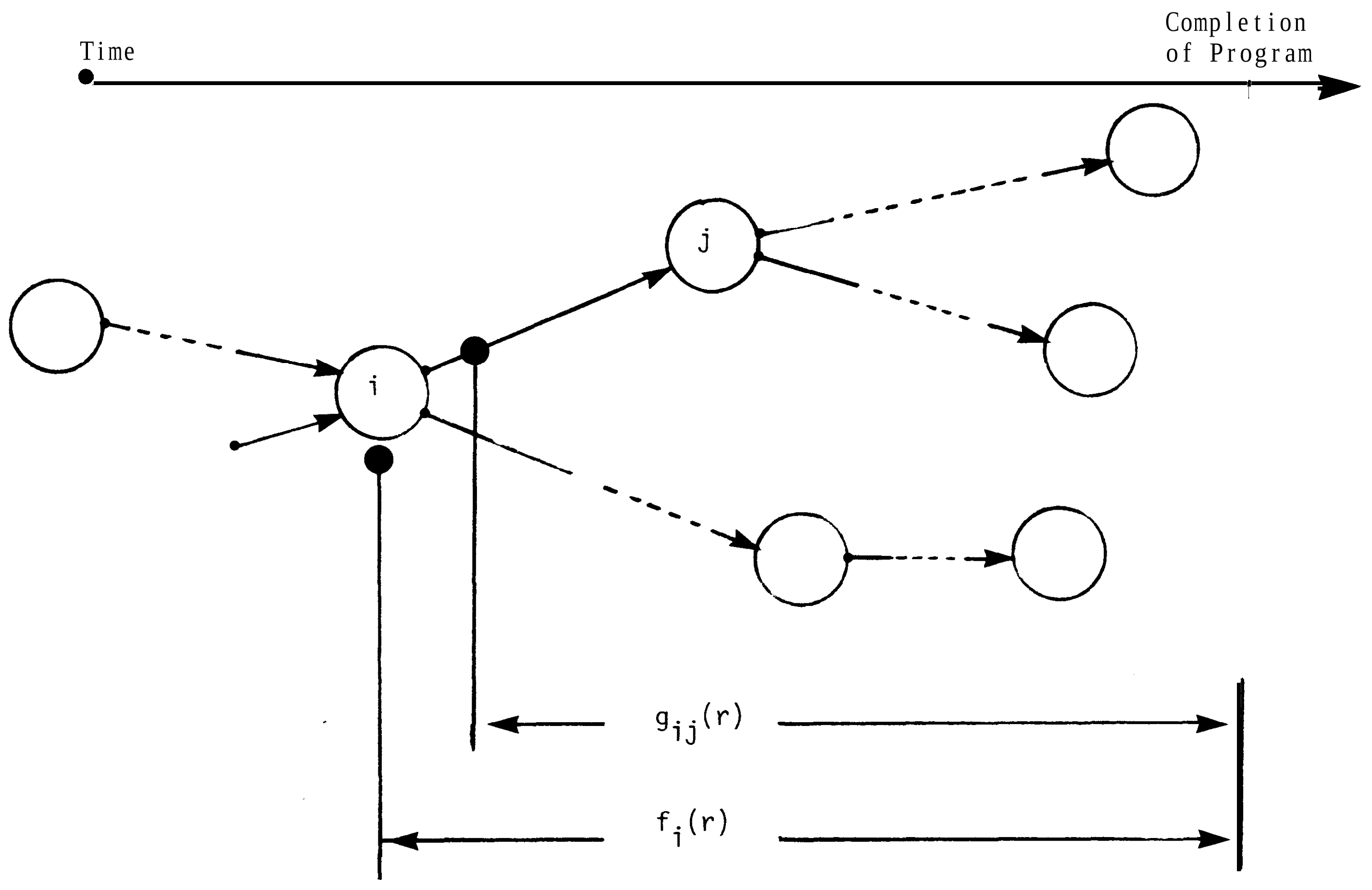


Figure 3.1 - THE AUXILIARY f(r) AND g(r) FUNCTIONS

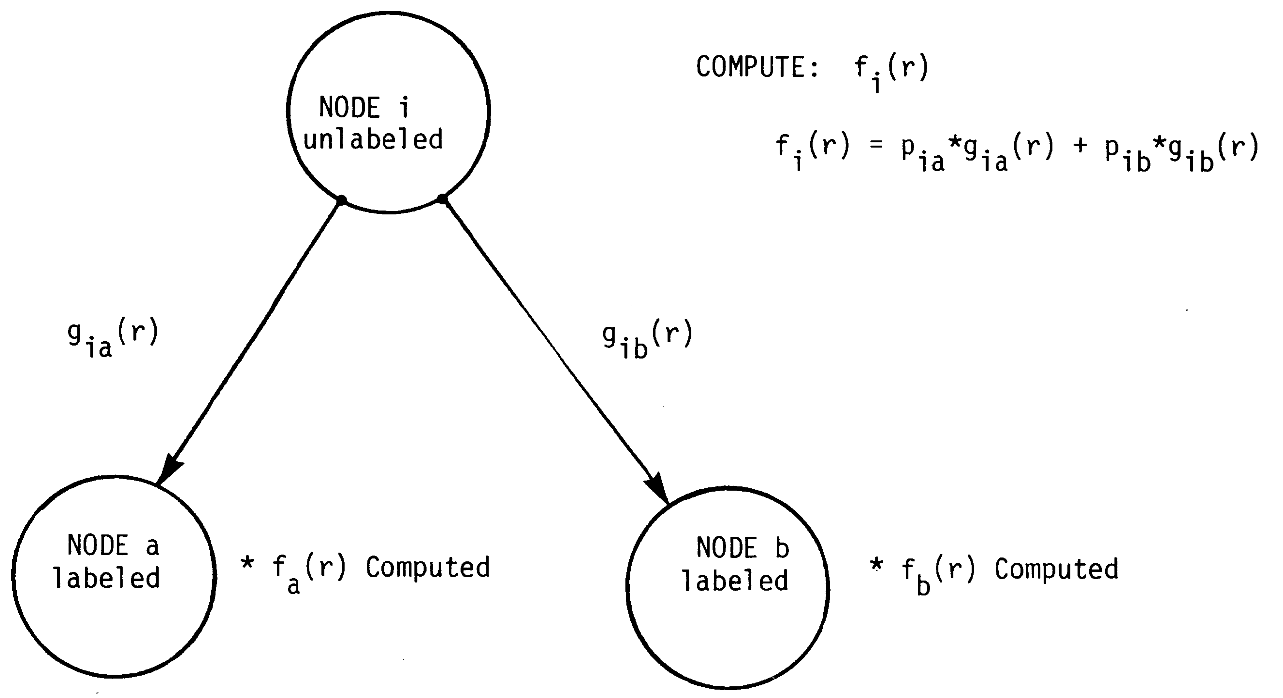


Figure 3.2 COMPUTATION OF THE MERT ALGORITHM

1. First, for each *labeled* node j which succeeds node i (i.e., for each edge (i,j) emanating from node i) define the function $g_{ij}(r)$ to be:

$$g_{ij}(r) = S_{ij} + E[t_j] + f_j(E[t_j] + L_{ij}) \qquad \text{if } r+E[t_j] > M$$

$$= E[t_j] + \min\{ S_{ij} + f_j(E[t_j] + L_{ij}), \; f_j(r + E[t_j]) \} \qquad \text{if } r+E[t_j] \le M.$$

equation (1)

2. Second, compute the optimal decision parameter B_{ij} for this edge (i,j). B_{ij} is defined to be the ***maximum*** value of the expected recovery time **r,** for which:

$$g_{ij}(r) = f_j(r + E[t_j]) + E[t_j]. \qquad \textbf{equation (2)}$$

3. Third, after $g_{ij}(r)$ has been computed for all edges (i,j), compute $f_i(r)$:

$$f_i(r) = \sum_{j \in (i,j)} p_{ij} \cdot g_{ij}(r), \qquad \textbf{equation (3)}$$

and *label* node i, indicating that the function **fi(r)** has been determined for this node.

Note that if **fi(r) =** ∞ for all **r** ≥ 0, then we are not able to meet the constraint of bounding the expected recovery time by the maximum value of **M** time units for this chosen graph formulation of the program, **G.**

Repeat the next (k + 1st) step of the algorithm.

3.5 Lemma: Termination of Algorithm

The MERT algorithm terminates only after all nodes have been labeled.

Proof (by contradiction)

Assume that the algorithm has terminated, leaving an unlabeled node P. If the algorithm terminated with an unlabeled node P, then there must exist a subgraph of successors to node P, ***all*** of whose nodes are ***unlabeled.*** Since all subgraphs of acyclic directed graphs are also acyclic, we can follow this path to an exit node. But all of the exit nodes were labeled on the 0th step of the algorithm. Therefore, by contradiction, the algorithm could not have terminated.

3.6 Lemma: Bounds on Termination

The MERT algorithm terminates within n steps if there are n nodes in the program graph C.

Proof

Each step if the algorithm will remove at least one unlabeled node from the program graph ***G.*** By ***Lemma*** 3.5, the algorithm terminates when all nodes have been labeled. Therefore it terminates within n steps.

3.7 Proof that the MERT Algorithm Minimizes the Expected Run Time

Proof by induction, that on the k^{th} step of the algorithm, the value determined for B_{ij} is the optimal decision parameter described previously in Section 2.5.

3.7.1 Case k=1

For each node i which is labeled on the first step, we have the following situation of Fig. 3.3, in which all nodes j, such that the edge (i,j) exists, are exit nodes.

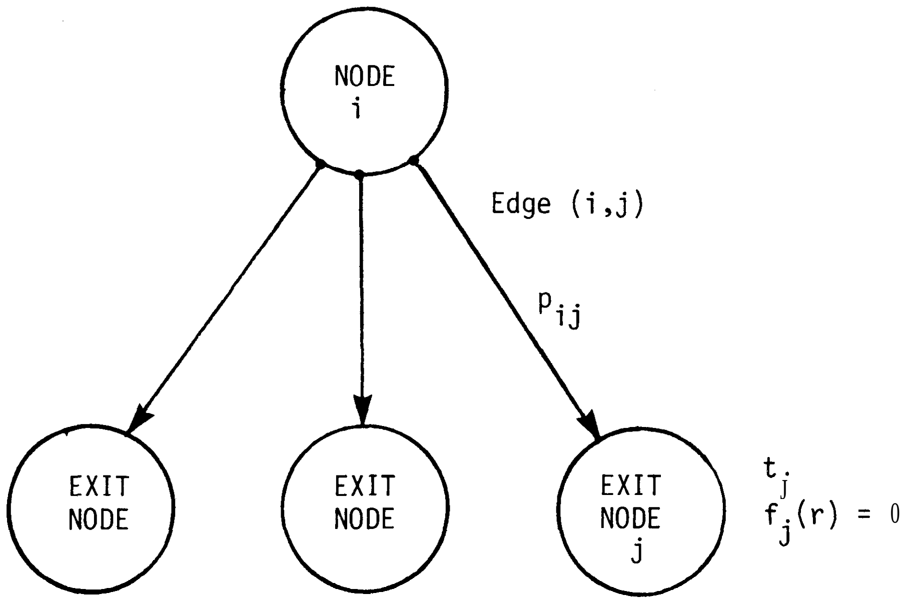


Figure 3.3 - MERT ALGORITHM, CASE $k=1$

Since $f_j(r) = 0$ for all of these exit nodes j, equation (1) reduces to:

$$g_{ij}(r) = S_{ij} + E[t_j] \qquad \text{if } r+E[t_j] > M$$

$$= E[t_j] \qquad \text{if } r+E[t_j] \leq M$$

Subcase: $r + E[t_j] > M$

If $r + E[t_j] > M$, a recovery checkpoint must be inserted on edge (i,j), otherwise the recovery time after task j completes, $r + E[t_j]$ may possibly exceed the maximum bound of M time units. If a rollback point must be inserted on edge (i,j), the minimum expected execution time $g_{ij}(r)$ at this point, *after* the completion of task i, until the end of the program, is the sum of:

1. The expected task execution time for task j, established previously in Section 3.1.2: $E[t_j]$.

2. The time taken to insert the rollback point on edge (i,j): S_{ij}. This includes saving the program state and those action which the data set interface processes takes upon receipt of the MARK command.

Thus,

$$g_{ij}(r) = E[t_j] + S_{ij}.$$

Subcase: $r + E[t_j] < M$

If $r + E[t_j] \leq M$, a recovery checkpoint need not be inserted on edge (i,j). Since node j is an exit node, the minimum expected execution time $g_{ij}(r)$ is the expected execution time for task j:

$$g_{ij}(r) = E[t_j].$$

$\underline{B_{ij}}$

The optimal decision, B_{ij}, for this case (k=1) when node j is an *exit* node is to *not* insert a rollback point on edge (i,j) as long as:

$$r + E[t_j] \leq M$$

Thus, B_{ij} for this particular edge (i,j) is:

$$B_{ij} = E[t_j]$$

Note also that B_{ij} is the maximum value of r for which:

$$g_{ij}(r) = f_j(r + E[t_j]) + E[t_j].$$

Computation of $f_i(r)$

The minimum expected time spent in program execution, after task i completes until the termination of the program, is then the weighted sum of the $g_{ij}(r)$ averaged over *all* program branches (i,j) emanating from this node i:

$$f_i(r) = \sum_{j \in (i,j)} p_{ij} \cdot g_{ij}(r)$$

Thus, the MERT algorithm is true for the case k = 1.

3.7.2 Induction Step

Assume that the algorithm is true for k = 1,2, . . . x-l. Prove it to be true for the case k = x.

At this point we are at an arbitrary edge (i,j) of the program graph G, as illustrated in **Fig. 3.4.**

If a recovery checkpoint is inserted on edge (i,j) immediately before the execution of task j, then the expected recovery time after task j completes is:

$$E[t_j] + L_{ij}.$$

If the expected recovery time after task j terminates is $E[t_j] + L_{ij}$, then by the induction hypothesis, the minimum expected execution time after task j completes, until the end of the program is:

$$f_j(E[t_j] + L_{ij})$$

Thus the total minimum expected run time, after task i completes and if task j is executed next, is the sum of:

1. the time taken to establish the recovery checkpoint, S_{ij}, plus

2. the expected time to execute task j, $E[t_j]$ plus

3. the minimum expected execution time after task j completes, $f_j(E[t_j] + L_{ij})$.

so:

$$g_{ij}(r) = S_{ij} + E[t_j] + f_j(E[t_j] + L_{ij})$$

If a recovery checkpoint is *not* inserted on edge (i,j), the expected recovery time immediately after task j terminates is:

$$r + E[t_j]$$

If the expected recovery time is $r + E[t_j]$, then by the induction hypothesis, the minimum expected execution time after the completion of task j is:

$$f_j(r + E[t_j]),$$

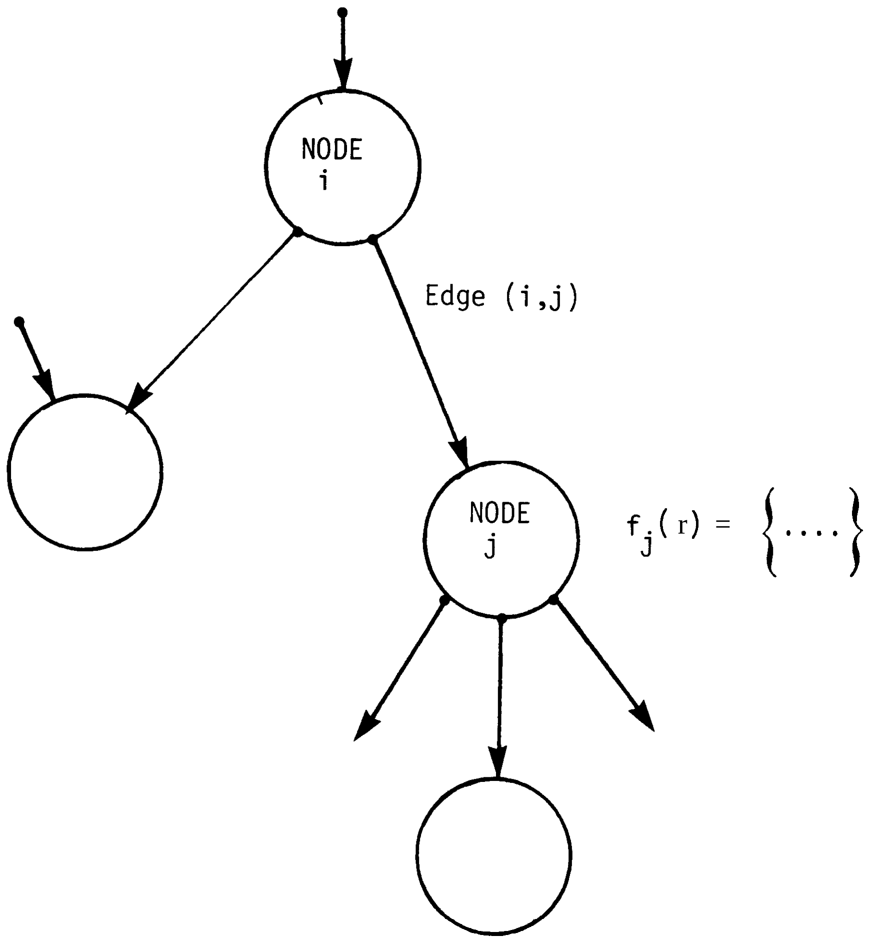


Figure 3.4 - MERT ALGORITHM, INDUCTION STEP

The total minimum expected run time after task i 'completes, if task j is executed next, is the expected time taken to execute task j plus the minimum expected execution time after task j completes:

$$g_{ij}(r) = E[t_j] + f_j(r + E[t_j]).$$

Subcase: $r + E[t_j] > M$

If $r + E[t_j] > M$, a rollback point must be inserted on edge (i,j) if the expected recovery time after task j is constrained not to exceed our bound of **M** time units. Thus:

$$g_{ij}(r) = S_{ij} + E[t_j] + f_j(E[t_j] + L_{ij})$$

Subcase: $r + E[t_j] \leq M$

If $r + E[t_j] < M$, then the option exists of not inserting a rollback point on edge (i,j). If it is not inserted, the minimum expected run time of the program is:

$$E[tj\,] + f_j(r + E[t_j])$$

If a rollback point is inserted, the minimum expected run time is:

$$S_{ij} + E[t_j] + f_j(E[t_j] + L_{ij})$$

So, picking the method which minimizes the expected execution time:

$$g_{ij}(r) = E[t_j] + \min \{ Sij + f_j(E[t_j] + L_{ij}), \\ f_j(r + E[t_j]) \}$$

$\underline{B_{ij}}$

The optimal decision for the edge (i,j) is to not insert the recovery checkpoint as long as the minimum expected execution time after processing task j until the end of the program:

$$f_j(r+E[t_j])$$

plus the expected execution time for task j:

$$E[t_j]$$

equals the minimum expected execution time after the processing of task i, along the (i,j) program branch:

$$g_{ij}(r)$$

B_{ij} then is that maximum value of r, below which:

$$g_{ij}(r) = f_j(r + E[t_j]) + E[t_j].$$

This is the value of B_{ij} which minimizes the expected execution time.

<u>Computation of $f_i(r)$</u>

Again

$$f_i(r) = \sum_{j \in (i,j)} p_{ij} \cdot g_{ij}(r),$$

$f_i(r)$ is the expected run time of the program after task i completes, averaged over all branches (i,j) emanating from node i.

Q.E.D.

Chapter 4

EXAMPLE ILLUSTRATING THE DYNAMIC INSERTION OF RECOVERY CHECKPOINTS

This chapter will attempt to illustrate the use and implementation of the previously developed MERT algorithm. A graph model of a ***typical*** sample program segment is analyzed by the MERT algorithm, and the static ***optimal decision*** *parameter set* $\{B_{ij}\}$ is computed. Then, using the recovery checkpoint insertion procedure of Section 2.5, the run-time behavior for this program graph is examined for varying conditions of execution parameters and error conditions.

4.1 Graph Model of a Typical Program

A graph model of the typical program is shown in Fig. 4.1. There are seven distinct tasks in the program. For simplicity, the Load and Save time for all branches (L_{ij}, S_{ij}) are are fixed for this example at L_{ij} = 3, S_{ij} = 4 for all branches (i,j). Any iterations (i.e., *FOR*, ***WHILE***, or *DO* Loops) in the program have been coalesced into a sequence of statements contained in one task. The other parameters for this program are:

t_i - The expected time to execute task i correctly (Section 2.2.2).

y_i - The expected time between the initiation of task i, and the detection of an error, if one occurs (Section 3.2.2).

q_i - The probability of encountering at least one error during the execution of task i (Section 3.2.1).

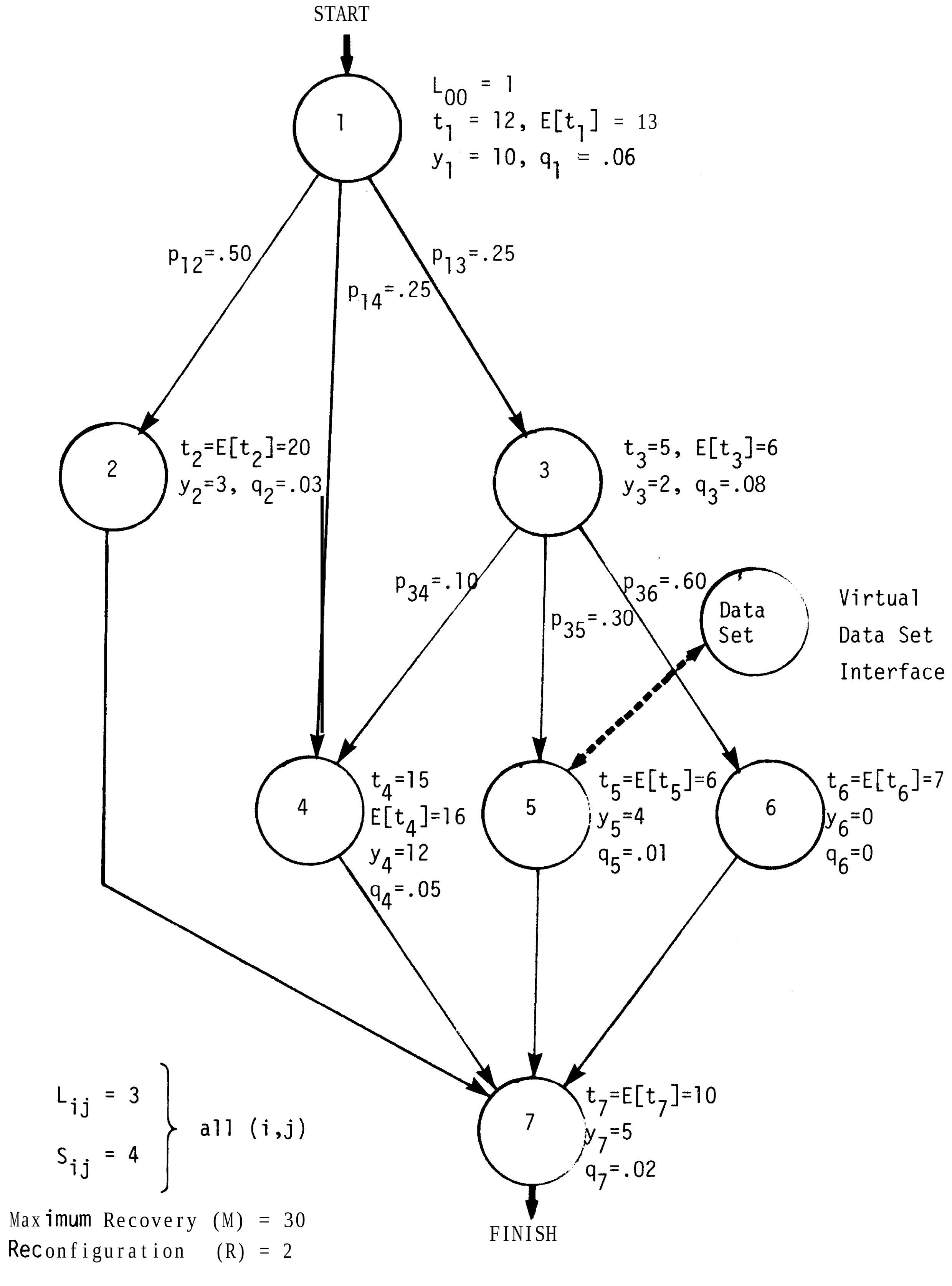


Figure 4.1 - GRAPH MODEL OF SAMPLE PROGRAM

p_{ij} - The probability (fixed and independent) that task j will follow **task i** (Section 2.2.4).

Also, assume that task 5 issues output to a lineprinter type data set (the virtual data set interface process described in Section 2.3.3).

The maximum expected recovery time, M, is 30 time units duration, and the system reconfiguration time, R, is 2 time units. The values given in **Fig. 4.1** for $E[t_i]$ (the expected task execution time for task i) are computed (Appendix A) from:

$E[t_i] = t_i + \{(R + y_i)\ q_i\} / (1-q_i)$ (Section 3.3)

4.2 Computation of the Optimal Decision Parameter Set

The optimal decision parameter set $\{B_{ij}\}$ for the program represented by **Fig. 4.1** is now obtained by application of the MERT algorithm which was presented in Section **3.4.2.**

The MERT algorithm has been coded as a BCPL program in Appendix A. The input data to this analysis program for our sample program is shown in Appendix B, and the output results obtained by the MERT analysis (the **fi(r),** $g_{ij}(r)$, and the set $\{B_{ij}\}$) are given in Appendix C.

Applying the MERT algorithm (Section 3.4.2) to the sample program graph in **Fig. 4.1.**

Step 0

On this initial step of the algorithm we define **fi(r)** $= 0$, for $r \leq M$ for all exit nodes of the program graph. In our sample program there is only one exit node, node 7. So:

$$f_7(r) = \begin{cases} 0, & \text{for } 0<r<30, \\ \infty, & \text{for } 30<r. \end{cases}$$

and mark node 7 as being labeled (i.e., $f_7(r)$ has been determined).

Step **1**

Node 7 is now labeled, and nodes 2, 4, 5 and 6 are unlabeled, having had all their successors labeled. Compute $g_{27}(r)$ and B_{27} from equations (1) and (2).

$$g_{27}(r) \begin{cases} = S27 + E[t_7] + f_7(E[t_7] + L_{27}) & \text{if } r+E[t_7]>M, \\ = E[t_7] + \mathit{min}\ \{S_{27} + f_7(E[t_7] + L_{27}), \\ \qquad f_7(r + E[t_7])\} & \text{if } r+E[t_7]\leq M. \end{cases}$$

substituting into equation (I).

If $r+E[t_7]>M$ (i.e., $r>20$) then:

$$S27 + E[t_7] + f_7(E[t_7] + L_{27}) = 4 + 10 + f_7(10+3)$$
$$= 14 + f_7(13) = 14.$$

If $r+E[t_7]\leq M$ (i.e., $r\leq 20$) then:

$$E[t_7] + \mathit{min}\ \{S_{27} + f_7(E[t_7] + L_{27}), \qquad f_7(r + E[t_7])\}$$
$$= 10 + \mathit{min}\ (4 + f_7(13), f_7(r + 10)\}$$
$$= 10 + \mathit{min}\ \{4,0\} = 10$$

so:

$$g_{27}(r) = \begin{cases} 10, & \text{for } 0<r\leq 20, \\ 14, & \text{for } 20<r\leq 30. \end{cases}$$

Note that $g_{27}(r) = f_7(r+E[t_7]) + E[t_7] = f_7(r+10) + 10$ for all $r\leq 20$, so using equation (2) we find our first optimal decision parameter:

B27 = 20.

Since there is only one edge emanating from node 2, we can compute $f_2(r)$ from equation (3) as:

$$f_2(r) = p_{27}\ g_{27}(r) = g_{27}(r).$$

So:

$$f_2(r) = \begin{cases} 10, & \text{for } 0<r\leq 20, \\ 14, & \text{for } 20<r\leq 30, \\ \infty, & \text{for } 30<r. \end{cases}$$

and label node 2, indicating that $f_2(r)$ has been determined.

Steps 2, 3, 4

Now compute $g_{47}(r)$, $g_{57}(r)$, $g_{67}(r)$ and B_{47}, B_{57}, and B_{67}. Using the same procedure as above, we determine that:

$$g_{47}(r) = g_{57}(r) = g_{67}(r) = \begin{cases} 10, & \text{for } 0<r\leq 20, \\ 14, & \text{for } 20<r\leq 30. \end{cases}$$

and:

$$B_{47} = B_{57} = B_{67} = 20.$$

Again, since there is only one edge leaving nodes 4, 5 and 6, we can compute $f_4(r)$, $f_5(r)$ and $f_6(r)$:

$$f_4(r) = f_5(r) = f_6(r) = \begin{cases} 10, & \text{for } 0<r\leq 20, \\ 14, & \text{for } 20<r\leq 30, \\ \infty, & \text{for } 30<r. \end{cases}$$

and label nodes 4, 5 and 6.

Step

Now, since node 3 is unlabeled, and all its successor nodes (4, 5 and 6) are labeled, we can compute $g_{34}(r)$ and B_{34} from equations (1) and (2):

$$g_{34}(r) \begin{cases} = s34 + E[t_4] + f_4(E[t_4] + L_{34}) & \text{if } r+E[t_4]>M, \\ = E[t_4] + \min\{S_{34} + f_4(E[t_4] + L_{34}), \\ \qquad f_4(r + E[t_4])\} & \text{if } r+E[t_4]\leq M. \end{cases}$$

If $r+E[t_4]>M$ then:

$$S_{34} + E[t_4] + f_4(E[t_4] + L_{34}) = 4 + 16 + f_4(16+3)$$
$$= 20 + f_4(19) = 20 + 10 = 30$$

If $r+E[t_4]\leq M$ (i.e., $r\leq 4$) then:

$$f_4(r + E[t_4]) = f_4(r+16) = 10 \qquad \text{for } r\leq 4$$

so:

$$g_{34}(r) = \begin{cases} 26, & \text{for } 0<r\leq 4, \\ 30, & \text{for } 4<r\leq 30. \end{cases}$$

Now note that $g_{34}(r) = f_4(r+E[t_4]) + E[t_4] = f_4(r+16) + 16$ for all $r\leq 14$, so using equation (2) we find that:

$$B_{34} = 14 .$$

Now compute $g_{35}(r)$, and B_{35} . Using the same method as above, we find that:

$$g_{35}(r) = \begin{cases} 16, & \text{for } 0<r\leq 14, \\ 20, & \text{for } 14<r\leq 30. \end{cases}$$

and:

$$B_{35} = 24.$$

And for $g_{36}(r)$ and B_{36}:

$$g_{36}(r) = \begin{cases} 17, & \text{for } 0<r\leq 13, \\ 21, & \text{for } 13<r\leq 30. \end{cases}$$

and:

$$B_{36} = 23.$$

Now, since $g_{ij}(r)$ has been computed for all edges leaving node 3 (i.e., $g_{34}(r)$, $g_{35}(r)$ and $g_{36}(r)$), we can use equation (3) to compute $f_3(r)$:

$$f_3(r) = \sum_{j \in \{4,5,6\}} p_{ij} \cdot g_{ij}(r) \qquad \text{equation (3)}$$

where $p_{34} = .10$, $p_{35} = .30$ and p36 = .60, and $g_{ij}(r)$ are those given above., This computation yields (See Fig. 4.2 for a pictorial representation of this calculation) :

$$f_3(r) = \begin{cases} 17, & \text{for } 0<r\leq 4, \\ 18, & \text{for } 4<r\leq 13, \\ 20, & \text{for } 13<r\leq 14, \\ 21, & \text{for } 14<r\leq 30, \\ \infty, & \text{for } 30<r. \end{cases}$$

and label node 3, indicating that $f_3(r)$ has been computed.

Step 6

Now compute $g_{12}(r)$, $g_{13}(r)$, $g_{14}(r)$ and B_{12}, **B13, and** B_{14}. **Using** the same procedures, we find that:

$$g_{12}(r) = \begin{cases} 34, & \text{for } 0<r\leq 10, \\ 38, & \text{for } 10<r\leq 30. \end{cases}$$

and:

$$B_{12} = 10.$$

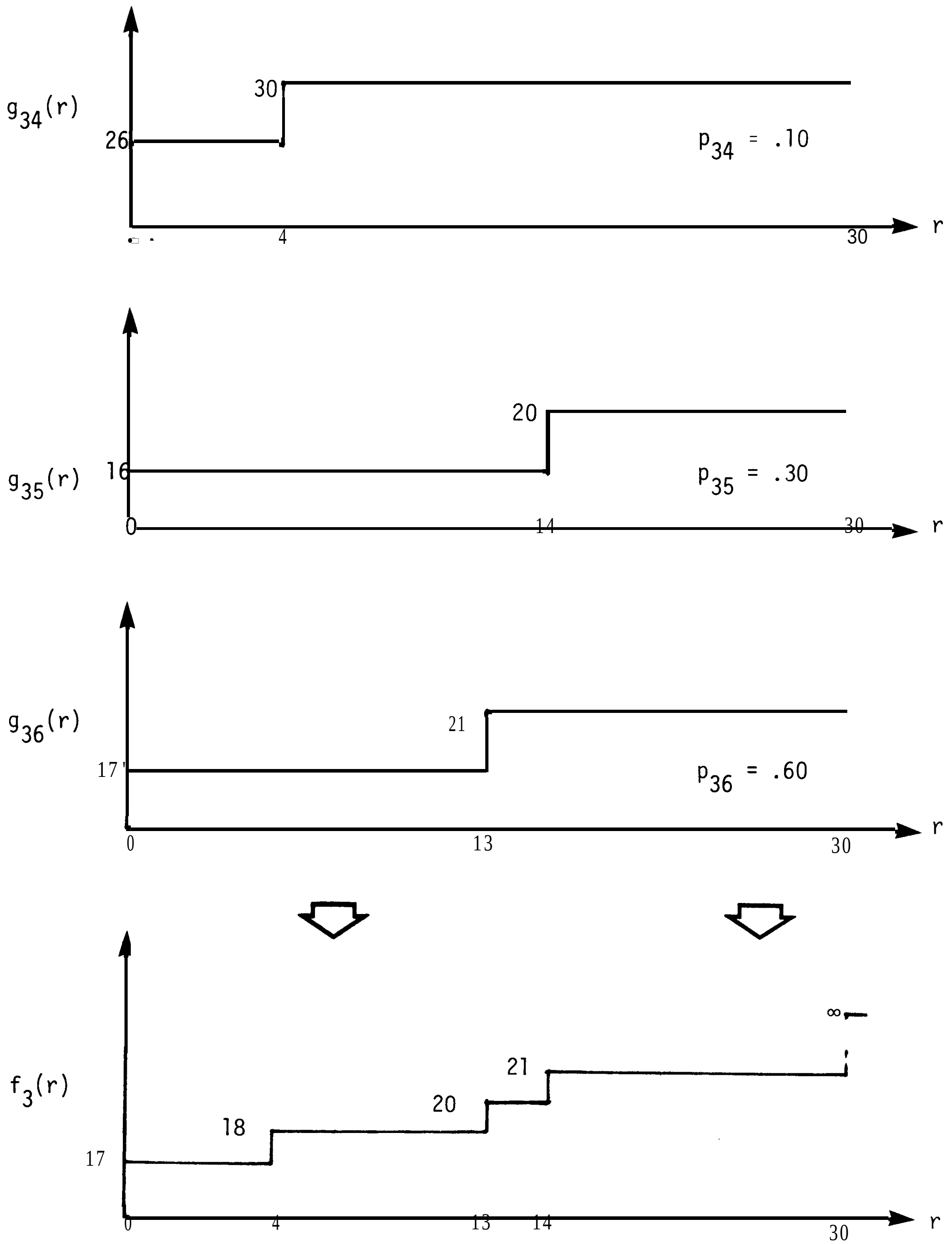


Figure 4.2 - CALCULATION OF $f_3(r)$ FOR SAMPLE PROGRAM

$$g_{13}(r) = \begin{cases} 24, & \text{for } 0<r\leq 7, \\ 26, & \text{for } 7<r\leq 8, \\ 27, & \text{for } 8<r\leq 24, \\ 28, & \text{for } 24<r\leq 30. \end{cases}$$

and:

$$B_{13} = 24 .$$

$$g_{14}(r) = \begin{cases} 26, & \text{for } 0<r\leq 4, \\ 30, & \text{for } 4<r\leq 30. \end{cases}$$

and:

$$B_{14} = 14.$$

Now, since $g_{12}(r)$, $g_{13}(r)$ and $g_{14}(r)$ have been computed, we can calculate $f_1(r)$. Using equation (3) we find that:

$$f_1(r) = \begin{cases} 29, & \text{for } 0<r\leq 4, \\ 30, & \text{for } 4<r\leq 7, \\ 31, & \text{for } 7<r\leq 10, \\ 33, & \text{for } 10<r\leq 30, \\ \infty, & \text{for } 30<r. \end{cases}$$

and label node 1.

Step 7

On this final step of the MERT algorithm we find that all of the nodes are labeled. The algorithm terminates, having computed the complete optimal decision parameter set $\{B_{ij}\}$.

These results are summarized in **Fig. 4.3.**

4.3 Example Execution of Sample Program

In the following examples, suppose that the given run of the sample program (represented by **Fig. 4.1)** executes tasks 1, 3, 5 and 7 sequentially. Also, assume that the program is initially loaded in time $L_{00} = 1$.

From $f_1(r)$ (Section 4.2) we find that the minimum total expected execution time for the program, including the time for placement of any required recovery checkpoints is:

$$E[t_1] + f_1(E[t_1] + L_{00}) = 13 + f_1(13 + 1) = 13 + 33 = 46.$$

The following examples make use of the checkpoint insertion algorithm presented in Section 2.5 (and **Fig. 2.6).**

4.3.1 Example 1

Suppose that in this given run of the program that the task execution times are:

OPTIMAL DECISION PARAMETER SET	
B_{12}	10
B_{13}	24
B_{14}	14
B_{27}	20
B_{47}	20
B_{57}	20
B_{67}	20
B_{34}	14
B_{35}	24
B_{36}	23

Figure 4.3 - TABLE OF MERT COMPUTED OPTIMAL DECISION PARAMETERS FOR SAMPLE PROGRAM

task 1 = 1

task 3 = 2

task 5 = 3

task 7 = 5

After task 1 completes, we interrogate the recovery time, r:

$$r = R + SysClock_1 - CPClock\ 1 + Load1\ (=L_{00})$$
$$= 2 + (1 - 0) + 1 = 4$$

and $r \leq M - B13 = 30 - 24 = 6$, so we do not insert a recovery checkpoint. Process task 3.

After task 3 completes, again interrogate **r:**

$$r = 2 + (1 + 2) + 1 = 6$$

$r \leq M - B_{35} = 30 - 24 = 6$, so no recovery checkpoint is placed. Process task 5.

After task 5 completes, interrogate **r:**

$$r = 2 + (1 + 2 + 3) + 1 = 9$$

and $r \leq M - B_{57} = 30 - 20 = 10$, so go on to process task 7.

Thus, for the execution characteristics of this particular example, it was not necessary to insert any recovery checkpoints.

4.3.2 Example 2

On this run of the program, the task execution times are:

task 1 = 2

task 3 = 5

task 5 = 4

task 7 = 5

After task 1:

$$r = 2 + (2 - 0) + 1 = 5$$

$r \leq M - B_{13} = 30 - 24 = 6$, so go on to process task 3. After task 3 completes, again interrogate r:

$$r = 2 + (2 + 5) + 1 = 10$$

$r > M - B_{35} = 30 - 24 = 6$, so we insert a recovery checkpoint on edge (3,5) (which consumes $S_{35} = 4$ time units). Now process task 5. After task 5 completes, interrogate **r:**

$$r = R + 4 + L_{35} = 2 + 4 + 3 = 9$$

$r \leq M - B_{57} = 30 - 20 = 10$, so go on to process task 7.

Thus, in example 2, one dynamic recovery checkpoint was placed between the execution of task 3 and task 5.

4.3.3 Example 3

On this run of the program, let the task execution times remain the same as above (Example 2):

task 1 = 2

task 3 = 5

task 5 = 4

task 7 = 5

But on this run, an error is detected during the processing of task 5.

When the error is detected, we roll back to the previously placed checkpoint which preceded task 5, (i.e., the one which was placed on edge (3,5), and backup the virtual data set which interfaces with task 5 (via the **RESTORE** command which will cause the output issued by task 5 to be revoked). This reloading process takes:

$$\mathbf{R + L_{35} (= reload_{ij} + T_{restore}) = 2 + 3 = 5 \text{ time units.}}$$

Assuming that the system reconfiguration **(R)** fixed the faulty module which caused task 5 to fail, ***and*** that this reloaded checkpoint contained the correct state information, we find that task 5 now executes successfully. After task 5 finishes:

$$\mathbf{r = R + 4 + L_{35} = 2 + 4 + 3 = 9}$$

and $\mathbf{r} \leq M - B_{57} = 10$, so continue on to process task 7.

4.4 Summary

In this chapter, the computation of the set $\{B_{ij}\}$ and its execution time use have been demonstrated. It has been shown that the optimal insertion of recovery checkpoints is a dynamic procedure, and is a function of the runtime characteristics - input parameters, actual task execution times, etc. - of the program. Given that the set $\{B_{ij}\}$ has been determined, the insertion of recovery checkpoints requires only a minimal runtime computational overhead.

This thesis has described a recovery method which guarantees that a computer system and its asociated data sets will be restored to an operational and consistent state within a given amount of time, minimizing the total overhead cost of creating recovery checkpoints.

BIBLIOGRAPHY

[1] S.M. Ornstein, W.R. Crowther, M.F. Kraley, R.D. Bressler, A. Michel, and F.E. Heart, "Pluribus - A Reliable Multiprocessor," Natl. Comp. Conf. 1975, pp. 551-559.

[2] W.C. Carter, D.C. Jessep, W.G. Bouricius, A.B. Wadia, C.E. McCarthy, and F.G. Milligan, "Design Techniques for Modular Architecture for Reliable Computer Systems," IBM Report RA12, March 1970.

[3] IBM Corporation, IBM OS Advanced Checkpoint/Restart, IBM Manual GC28-6708, 1974.

[4] F.P. Mathur, and A. Avizienis, "Reliability Analysis of a Hybrid Redundant Digital System - Generalized TMR with Self-Repair," Spring Joint Comp. Conf. (AFIPS) 1970.

[5] J. Losq, "A Highly Efficient Redundancy Scheme: Self Purging Redundancy," IEEE Trans. on Computers, Vol. C-25, No. 6, June 1976, pp. 569-577

[6] W.C. Carter, and C.E. McCarthy, "Implementation of an Experimental Fault Tolerant Memory System, IEEE Trans. on Computers, Vol. C-25, No. 6, June 1976, pp. 557-568.

[7] A. Avizien is, "Design of Fault Tolerant Computers," Fall Joint Comp. Conf., AFIPS Press, New Jersey, 1967, pp. 733-743.

[8] B. Randell, "System Structure for Software Fault Tolerance," IEEE Trans. on Software Eng., Vol. SE-l, No. 2, June 1975, pp. 220-232.

[9] J. Von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," Automata Studies, pp. 43-98, Princeton University Press, Princeton, N.J., 1956.

[10] J.A. Rohr, "System Software for a Fault-Tolerant Digital Compter," Ph.D. Thesis, Computer Science Department, Univ. of Illinois, 1973.

[11] BBN Inc., Pluribus Document 1: Overview, BBN Report No. 2999, May 1975.

[12] D.C. Russell, "Error Recovery and Process Communication," Ph.D. Thesis, Computer Science Department, Stanford University, April 1976.

[13] K.M. Chandy and C.V. Ramamoorthy, "Rollback and Recovery Strategies for Computer Programs," IEEE Trans. on Computers, Vol. C-21, No. 6, June 1972, pp. 546-556.

[14] K.M. Chandy, "A Survey of Analytic Models of Rollback and Recovery Strategies," IEEE Computer Magazine, Vol. 9, No. 4, April 1976, pp. 40-47.

[15] C.V. Ramamoorthy, K.M. Chandy, and A.E. Cowan, "A Framework for Hardware - Software Tradeoffs in the design of Fault - Tolerant Corn pu ters," Proc. Fall Joint Comp. Conf., AFIPS Press, New Jersey, 1972, pp. 55-64.

[16] R.W. Floyd, "Assigning Meanings to Programs," Proc. Symp. Appl. Math., American Math. Soc., Vol. 19, pp. 19-32, 1967.

[17] C.A.R. Hoare, "Towards a Theory of Parallel Programming," In Operating Systems Techniques, Hoare and Perott, Academic Press, New York, 1972.

[18] H.C. Lauer, "Correctness in Operating Systems," Ph.D. Thesis, Carnegie-Mellon University, 1973.

[19] J.C. King, "Proving Programs to be Correct," IEEE Trans. on Computers, Vol. C-20, No. 11, November 1971, pp. 1331-1336.

[20] C.V. Ramamoorthy, K.M. Chandy, and M.J. Gonzalez, "Optimal Scheduling Strategies in a Multiprocessor System," IEEE Trans.

Computers, Vol. C-21, Feb. 1972, pp. 137-146.

[21] E.C. Russell, and G. Estrin, "Measurement Based Automatic Analysis of Fortran Programs," Spring Joint Comp. Conf., AFIPS Press, New Jersey, 1969.

[22] P.M. Merlin, "The Time-Petri-Net and the Recoverability of Processes," U.C., Irvine, Technical Report 48, May 1974.

[23] B. Beizer, "Analytical Techniques for the Statistical Evaluation of Program Running Time," Fall Joint Comp. Conf., AFIPS Press, New Jersey, 1970, pp. 519-525.

[24] W. Teitelman, "The Interlisp Editor," In Interlisp Reference Manual, Xerox Palo Alto Research Center, 1975.

Appendix A

THE MERT ANALYSIS PROGRAM

```
// Program for computation of MERT algorithm
// Input parameters stored on file: G.IN


get "STREAMS.D"              // Stream Definitions

external
       [            // system routines
       Gets
       keys
       Puts
       wss
       ws
       OpenFile
       Closes
       DeleteFile
       ]

manifest
       [
       nmax = 100 // max number of nodes
       emax = 100 // max number of edges
       fmax = 300 // breakpoint storage in function list
       infinity = #77777       // our 16-bit infinity
       1

static
       [
       N            // node list
       E            // edge list
       F            // function list
       diskin       // disk input file
       diskout      // disk output file
       nodes        // number of nodes
       edges        // number of edges
       M            // max recovery time
       R            // re-conf iguration time
       fnext        // free pointer into Function list chain
       1

structure string:
       [            // string template
       length byte
       char↑1,255 byte
```

```
        1

structure node:
        [           // each entry corresponds to one node in the graph
        indx ↑ 1,nmax:
                    [
                    count word // number of successors to this node
                    plink word // list of predecessors - pointer to
                                         // EDGE list
                    slink word // l ist of successors - pointer to
                                         // EDGE list
                    ti word      // expected running time of this node
                    Eti word     // expected task execution time - computed by DetEti
                    yi word      // expected time until detection of error
                    qi word      // probability of error occurring (per cent)
                    flink word // pointer to first element of the
                               // f function in FUNCTION list
                    labelled word          // this vertex is labelled
                    1
        1

structure edge:
        [           // each entry corresponds to one edge in the graph
        indxr 1,emax:
                    [
                    pred word   // predecessor node
                    plink word  // linked list (in EDGE) of
                                         // predecessors
                    succ word   // successor node
                    slink word  // linked list (in EDGE) of
                                         // successors
                    load word   // load time for this (pred => succ)
                                         // edge
                    save word   // save time for this (pred => succ)
                                         // edge
                    pij word    // probability of taking this program branch
                    glin k word // pointer to first element of the g
                                         // function in FUNCTION list
                    Bij word               // decision variable B(i,j)
                    1
        ]

structure   function:
        [           // each entry corresponds to a breakpoint of the
                    // f or g function
        indxt 1,fmax:
                    [
                    x word      // abscissa of discontinuity
                    y word      // ordinate of discontinuity
                    xylink word // points to the next breakpoint
                    ]
```

```
let Main () be
      [Main

      // Create the Node, Edge and Function storage areas
      let v = vec (size node)/16;            N = v
      let v = vec (size edge)/16;            E = v
      let v = vec (size function)/16;        F = v

      Ws ("*cStart of Program*c")           // Initialization on screen

      diskin = Open File ("G.IN", ksTypeReadOnly, charItem)
      if diskin eq 0 then
                  [           // Input file not on directory
                  Ws ("*cFile: G.IN not on disk")
                  goto FIN
                  ]
      DeleteFile ("G.OUT")
      diskout = OpenFile ("G.OUT", ksTypeReadWrite, charItem)


      Filllists (diskin, diskout)
      Printlists (diskout)
      Mert ()      // The MERT Algorithm
      Printlists (diskout)

FIN: Wrapup ()

      ]Main



and Mert () be
      [Mert

      // The MERT Algorithm - Chapter 3

      // initialize:  for all terminal nodes: F(r) = 0 for r le M

      for i = 1 to nodes do
                  [
                  if N>>node.indx↑i.count ne 0 then loop
                  N>>node.indx↑i.flink = fnext
                  Enterxy (M, infinity, 0) // store breakpoints
                  N>>node.indx↑i.labelled = true       // label exit node
                  PrintF (i, 0, N>>node.indx↑i.flink, diskout)
                  ]


LOOP:
      // look for an unlabelled vertex which has all of its successors
      // labelled

      let found = false
      for i = 1 to nodes do
```

```
[
if N > >node.indx ↑ i.labelled then loop
let successorlabelled = true
let ptr =  N>>node.indx↑i.slink
while ptr ne 0 do
          [
          let j = E> >edge.indx↑ptr.succ
          if not N>>node.indx↑j.labelled then
                    [
                    successorlabelled = false
                    break
                    ]
          ptr = E>>edge.indx↑ptr.slink
          ]

if not successorlabelled then loop

// at this point we know that all of the successors
// to node i are labelled

found = true
let ijptr = N>>node.indx↑i.slink
while ijptr ne 0 do
          [
          // for all edges j (i => j) compute Gij(r)
          let j = E>>edge.indx↑ijptr.succ

          Wss (diskout, "*cVertex i is unlabelled and vertex j is
labelled (i,j)= ")

          Wds (diskout, i)
          Wds (diskout, j)

          DetGij (i, j, ijptr)
          PrintF (i, j, E>>edge.indx↑ijptr.glink, diskout)


          // now set Bij
          E>>edge.indx↑ijptr.Bij = DetBij (i, j, ijptr)
          // print Bij
          Wss (diskout, "*cBij: (1, J, Bij) ")
          Wds (diskout, i)
          Wds (diskout, j)
          Wds (diskout, E>>edge.indx↑ijptr.Bij)

          ijptr = E>>edge.indx↑ijptr.slink
          ]

// now compute Fi( r)
let stop = Det Fi (i)

PrintF (i, 0, N>>node.indx↑i.flink, diskout)

// label node i to show that Fi(r) has been computed
```

```
                N > >node.indx↑i.labelled = true

                if stop then
                           [
                           Wss (diskout, "*cFi(r) = infinity for 0 le r le M")
                           Wss (diskout, "*cNode = ")
                           Wds (diskout, i)
                           return
                           ]

                ]

        // try to find another node whose successors are all labelled

        if found then goto LOOP

        ]Mert


and DetGij (i, j, ijptr) be
        [ DetGij

        // determine Gij(r)
        //
        // r+E[tj] gr M:          Gij(r) = Sij + E[tj] + fj(E[tj] + Lij)
        // r+E[tj] le M:                 = E[tj] + min { fj(r+E[tj]),
        //                                                Sij + fj(E[ tj] + Lij) }

        let Etj = N> >node.indx↑j.Eti
        let bp = M-Etj            // breakpoint
        let Lij = E>>edge.indx↑ijptr.load
        let Sij = E>>edge.indx↑ijptr.save

        let Fjptr = N>>node.indx↑j.flink
        let sl = Sij + Evalfg (Fjptr, Lij+Etj)

        // now construct Gij(r)

        let ysave = 0
        let sp = 0
        E>>edge.indx↑ijptr.glink = fnext


        for r = 0 to bp do
                [
                let yval = Evalfg (Fjptr, r + Etj)
                if yval gr sl then yval = sl
                if (yval + Etj) ne ysave then
                           [
                           if sp ne 0 then F>>function.indx↑sp.xylink = fnext
                           sp = fnext
                           Enterxy (r, yval + Etj, 0)
                           ysave = yval + Etj
                           ]
                ]

        // now enter breakpoint value if it isn't there
        if (sl + Etj) ne ysave then
                [
                if sp ne 0 then F>>function.indx↑sp.xylink = fnext
                sp = fnext
                Enterxy (bp, sl + Etj, 0)
                ]

        F>>function.indx↑sp.xylink = fnext
        Enterxy (M, infinity, 0)

        ]DetGij


and Det Fi (i) = valof
        [DetFi

        // Determine Fi(r) = SUM over all edges (i,j) of Gij(r)*pij, r le M

        let stop = false
        let ijptr = N>>node.indx↑i.slink
        N>>node.indx↑i.flink = fnext

        let ysave = 0
        let sp = 0
        for r = 0 to M do
                [rloop
                let infinflag = false
                let ysum = 0
                let flag = false
                let ijptr = N>>node.indx↑i.slink
                while ijptr ne 0 do
                        [               // pij * gij(r)
                        let Gijptr = E>>edge.indx↑ijptr.glink
                        let pij = E>>edge.indx↑ijptr.pij
                        let Gval = Evalfg (Gijptr, r)
                        if Gval eq infinity then infinflag = true
                        ysum = ysum + (Gval * pij)
                        ijptr = E> >edge.indx t ijptr.slink
                        ]
                ysum = ysum / 100       // Normalize:  pij in per cent

                if infinflag then ysum = infinity
                if ysave ne ysum then
                        [               // Enter this breakpoint into the function
list
                        if sp ne 0 then F>>function.indx↑sp.xylink = fnext
                        sp = fnext
                        Enterxy (r, ysum, 0)
                        ysave = ysum
                        ]
                if ysave eq infinity then
                          if r ls M then stop = true

                ]rloop

        resultis stop

        ]DetFi


    and DetBij (i, j, ijptr) = valof
        [ DetBij

        // Compute Bij such that:
        // gij(r) = fj( r + E[tj]) + E[tj]    for r le Bij

        let Etj = N>>node.indx↑j.Eti
        let Gijptr = E>>edge.indx↑ijptr.glink
        let Fjptr = N>>node.indx↑j.flink

        for r = 0 to M do
                [           // loop while Gij (r) = Fj (r + E[tj]) + E[tj]
                if (Evalfg (Fjptr, r + Etj) + Etj) ne
                          Evalfg (Gijptr, r) then resultis r
                ]

        resultis M

        ]DetBij


    and DetEti (i) be
        [ DetEti

        // Determine E[ti] for node i:
        //     E[ti] = ti + ( (R+yi)*qi / (1-qi) )

        let ti = N>>node.indx↑i.ti
        let yi = N>>node.indx↑i.yi
        let qi = N>>node.indx↑i.qi           // note that qi is in %

        let Eti = ti + (((R+yi)*qi) / (100 - qi))
        N>>node.indx↑i.Eti = Eti
        ]DetEti


    and Filllists (strmin, strmout) be
        [Filllists

        // create the EDGE and NODE lists from the graph model
        // input parameter file (strmin) on file: G.IN

        Wss (strmout, "*cEnter the Maximun Recovery Time M: ")
        M = Getnum (strmin, strmout)
        Wss (strmout, "*cEnter the Re-conf'iguration Time R: ")
        R = Getnum (strmin, strmout)
        Wss (strmout, "*cEnter the number of nodes in graph: ")
        nodes = Getnum (strmin, strmout)
        edges = 1

        for i = 1 to nodes do
                [iloop
                Wss (strmout, "*cEnter (Ti Yi Qi(%) for node ")
                Wds (strmout, i)
                Wss (strrnout, ": ")
                N>>node.indx↑i.ti = Getnum (strmin, strmout)
                N>>node.indx↑i.yi = Getnum (strmin, strmout)
                N>>node.indx↑i.qi = Getnum (strmin, strrnout)

                // Determine E[ti]
                DetEti (i)

                Wss (strmout, "For each successor to node ")
                Wds (strmout, i)
                Wss (strrnout, " enter:*c")
                Wss (strmout, "   (Successor Node, Load Time, Save Time, Pij(%)
)*c")
                let cnt = 0
                let tpij = 0
                while true do
                        [edgeloop
                        Wss (strrnout, "      ")
                        let succ = Getnum (strmin, strmout)
                        if succ eq 0 then
                                [
                                if tpij ne 100 then Wss (strmout,
                                 "*cSum of pij's neq 100 ******")
                                break
                                ]

                        cnt = cnt +1
                        E>>edge.indx↑edges.succ = succ
                        E>>edge.indx↑edges.pred = i
                        E>>edge.indx↑edges.load = Getnum (strmin, strmout)
                        E>>edge.indx↑edges.save = Getnrrm (strmin, strmout)
                        E>>edge.indx↑edges.pij = Getnum (strmin, strmout)
                        tpij = tpij + E>>edge.indx↑edges.pij
                        E>>edge.indx↑edges.glink = 0
                        E>>edge.indx↑edges.Bij = 0
                        test (cnt eq 1)
                                ifso E>>edge.indx↑edges.slink = 0
                                ifnot E>>edge.indx↑edges.slink = edges-l
                        edges = edges + 1
                        ]edgeloop

                test (cnt eq 0)
                        ifso N>>node.indx↑i.slink = 0
```

```
                    ifnot N>>node.indx↑i.slink = edges - 1
            N >>node.indx↑ i.count = cnt
            N>>node.indx↑i.flink = 0
            N>>node.indx↑i.labelled = false
            ]iloop

      edges = edges - 1

      // create the predecessor linked list

      for i = 1 to nodes do
            [iloop
            let ptr = 0
            for j = 1 to edges do
                  [jloop
                  if E>>edge.indxtj.succ ne i then loop
                  test (ptr eq 0)
                        ifso E>>edge.indx↑j.plink = 0
                        ifnot E>>edge.indx↑j.plink = ptr

                  ptr = j
                  ]jloop
            N>>node.indx↑i.plink = ptr
            ]iloop

      // initialize Function list by linking the free pointer chain

      for i = 1 to fmax do
            [
            F>>function.indxti.xylink = i+l
            3
      F>>function.indx↑fmax.xylink = 0 // end of chain
      fnext = 1


      ]Filllists



and Printlists (strmout) be
      [Printlists

      // Print the contents of the Node list

      Wss (strmout,"*cNode List:*c")
      Wss (strmout,"    Node  Count  Plink    Slink")
      Wss (strmout,"    Ti    Yi    Qi-%  E[ti]  Flink    Labelled*c")
      for i = 1 to nodes do
            [iloop
            Wds (strmout,i)
            Wds (strmout,N>>node.indx↑i.count)
            Wds (strmout,N>>node.indx↑i.plink)
            Wss (strniout," ")
            Wds (strmout,N>>node.indx↑i.slink)
```

```
                    Wds (strmout,N>>node.indx↑i.ti)
                    Wds (strmout,N>>node.indx↑i.yi)
                    Wds (strmout,N>>node.indx↑i.qi)
                    Wds (strmout,N>>node.indx↑i.Eti)
                    Wds (strmout,N>>node.indx↑i.flink)
                    test N>>node.indx↑i.labelled
                              ifso Wss (strmout, "        true*c")
                              ifnot Wss (strniout, "       false*c")
                    ]iloop

          // Print the contents of the Edge list

          Wss (strniout,"*c*cEdge List:*c")
          Wss (strniout,"   Index   Pred  Plink  Succ ")
          Wss (strmout,"Slink Load Save Pij-% Glink   Bij*c")
          for j = 1 to edges do
                    [jloop
                    Wds (strmout,j)
                    Wds (strmout,E>>edge.indx↑j.pred)
                    Wds (strmout,E>>edge.indx↑j.plink)
                    Wds (strmout,E>>edge.indx↑j.succ)
                    Wds (strmout,E>>edge.indx↑j.slink)
                    Wds (strmout,E>>edge.indx↑j.load)
                    Wds (strmout,E>>edge.indx↑j.save)
                    Wds (strmout,E>>edge.indx↑j.pij)
                    Wds (strmout,E>>edge.indx↑j.glink)
                    Wds (strmout,E>>edge.indx↑j.Bij)
                    Wss (strmout,"*c")
                    ]jloop

          ]Printlists


and Evalfg (ptr, x) = valof
          [ Evalfg

          // evaluate the f or g function (whose first element is pointed
          // to by ptr) with argument x

          x = x+1
          let result = 0
          while ptr ne 0 do
                    [
                    if x le F>>function.indx↑ptr.x then break
                    result = F>>function.indx↑ptr.y
                    ptr = F>>function.indx↑ptr.xylink
                    ]
          resultis result
          ] Evalfg


and Enterxy (x, y, link) be
          [ Enterxy
```

```
        // make an (x,y) entry on the Function list

        let i = fnext
        fnext =  F>>function.indxti.xylink
        F> >f unction.indx ↑ i.x = x
        F> >f unction.indxt i.y = y
        F>>function.indx↑i.xylink = link
        if fnext eq 0 then
                    [
                    // no more space
                    Wss (diskout, "*cFunction list filled")
                    Gets (keys)
                    1
        ]Enterxy


and PrintF (i, j, ptr, strmout) be
        [PrintF

        // Print either Fi(r) or Gij(r)

        if j eq 0 then
                    [
                    Wss (strmout,"*cFunction - F")
                    Wds (strmout, i)
                    Wss (strmout,"(r) ")
                    1
        if j ne 0 then
                    [
                    Wss (strmout,"*cFunction - G")
                    Wds (strmout,i); Wds (strmout,j)
                    Wss (strmout,"(r) ")
                    ]

        while ptr ne 0 do
                    [
                    Wss (strmout, "*c      ")
                    Wds (strmout,F> >f unction.indx ↑ ptr.x)
                    let yval = F>>function.indx↑ptr.y
                    test yval eq infinity
                                ifsO Wss (strmout, "   inf")
                                ifnot Wds (strmout, yval)
                    ptr = F>>function.indx↑ptr.xylink
                    1
        Wss (strmout, "*c")

        ]PrintF


and Getnum (strmin, strmout) = valof
        [Getnum

        // return a binary number from the keyboard
        let c = 0; let n = 0
        while true do
                    [
                    c = Gets (strmin)
                    Puts (strmout,c)
                    if c eq #40 % c eq #15 then resultis n
                    n = n*10 + c-$0
                    ]
        ]Getnum


and Wds (strm, val) be
        [Wds

        // Write decimal value: val to stream: strm

        let outstr = vec 5
        for i = 7 to 1 by -1 do
                    [
                    outstr>>string.char↑i = (val rem 10) + $0
                    val = val / 10
                    ]
        for i = 1 to 6 do
                    [
                    if outstr>>string.char↑i ne $0 then break
                    outstr>>string.char↑i = $
                    ]
        outstr>>string.length = 7
        Wss (strm, outstr)
        ]Wds


and Wrapup () be
        [Wrapup

        // Close disk files, etc.

        Closes (diskin)
        Closes (diskout)
        Ws ("*cEnd of Program")
        finish
        ]Wrapup
```

Appendix B

SAMPLE INPUT DATA FOR MERT ANALYSIS PROGRAM

```
Enter the Maximun Recovery Time M: 30

Enter the Re-configuration Time R: 2

Enter the number of nodes in graph: 7

Enter (t_i y_i q_i(%) for node      1:  12 10 6
For each successor to node          1 enter:
   (Successor Node, Load time, Save time, p_ij(%) )
    2 3 4 50
    3 3 4 25
    4 3 4 25


Enter (t_i y_i q_i(%) for node      2:  20 3 3
For each successor to node          2 enter:
   (Successor Node, Load time, Save time, p_ij(%) )
    7 3 4 100


Enter (t_i y_i q_i(%) for node      3:  5 2 8
For each successor to node          3 enter:
   (Successor Node, Load time, Save time, p_ij(%) )
    4 3 4 10
    5 3 4 30
    6 3 4 60


Enter (t_i y_i q_i(%) for node      4:  15 12 5
For each successor to node          4 enter:
   (Successor Node, Load time, Save time, p_ij(%) )
    7 3 4 100
```

```
Enter (t_i y_i q_i(%) for node        5:  6 4 1
For each successor to node            5 enter:
   (Successor Node, Load time, Save time, p_ij(%) )
     7 3 4 100


Enter (t_i y_i q_i(%) for node        6:  7 0 0
For each successor to node            6 enter:
   (Successor Node, Load time, Save time, p_ij(%) )
     7 3 4 100


Enter (t_i y_i q_i(%) for node        7: 10 5 2
For each successor to node            7 enter:
   (Successor Node, Load time, Save time, p_ij(%) )
    <EOF>
```

Appendix C

OUTPUT DATA FROM MERT ANALYSIS PROGRAM

(Note: $f_i(r)$ and $g_{ij}(r)$ are step functions. They are stored by storing their breakpoint values. i.e., $f_7(r)$: 30 ∞ corresponds to:

$$f_7(r) = \begin{cases} 0, & for\ r \leq 30, \\ \infty & for\ r > 30.) \end{cases}$$

```
Function - f7(r)
        30    ∞

Vertex i is unlabelled and vertex j is labelled (i,j) = 2       7
Function - g27(r)
         0    10
        20    14
        30    ∞

Bij: (i, j, Bij)       2       7      20
Function - f2(r)
         0    10
        20    14
        30    ∞

Vertex i is unlabelled and vertex j is labelled (i,j) = 4       7
Function - g47(r)
         0    10
        20    14
        30    ∞

Bij: (i, j, Bij)       4       7      20
Function - f4(r)
         0    10
        20    14
        30    00

Vertex i is unlabelled and vertex j is labelled (i,j) = 5       7
Function - g57(r)
         0    10
        20    14
        30    ∞

Bij: (i, j, Bij)       5       7      20
Function - f5(r)
         0    10
        20    14
```

```
              30       ∞

Vertex  i  is  unlabelled  and  vertex  j  is  labelled  (i,j)  =  6        7
Function  -  g67(r)
               0      10
              20      14
              30      ∞

Bij: (i,  j,  Bij)          6         7         20
Function  -  f6(r)
               0      10
              20      14      .
              30      ∞

Vertex  i  is  unlabelled  and  vertex  j  is  labelled  (i,j)  =  3        6
Function  -  g36(r)
               0      17
              13      21
              30      ∞

Bij: (i,  j,  Bij)          3         6         23
Vertex  i  is  unlabelled  and  vertex  j  is  labelled  (i,j)  =  3        5
Function  -  g35(r)
               0      16
              14      20
              30      00

Bij: (i,  j,  Bij)          3         5         24
Vertex  i  is  unlabelled  and  vertex  j  is  labelled  (i,j)  =  3        4
Function  -  g34(r)
               0      26
               4      30
              30      00

Bij: (i,  j,  Bij)          3         4         14
Function  -  f3(r)
               0      17
               4      18
              13      20
              14      21
              30      00

Vertex  i  is  unlabelled  and  vertex  j  is  labelled  (i,j)  =  1        4
Function  -  g14(r)
               0      26
               4      30
              30      ∞

Bij: (i,  j,  Bij)          1         4         14
Vertex  i  is  unlabelled  and  vertex  j  is  labelled  (i,j)  =  1        3
Function  -  g13(r)
               0      24
               7      26
               8      27
```

24	28
30	∞

B_{ij}: (i, j, B_{ij}) 1 3 24
Vertex i is unlabelled and vertex j is labelled (i j) = 1 2
Function - $g_{12}(r)$

0	34
10	38
30	∞

B_{ij}: (i, j, B_{ij}) 1 2 10
Function - $f_1(r)$

0	29
4	30
7	31
10	33
30	∞