AD-A064 489 UNIVERSITY OF SOUTHERN CALIFORNIA LOS ANGELES F/G 9/2
METHODOLOGIES AND TOOLS FOR DEVELOPING ROBUST FTSC SOFTWARE.(U)
AUG 78 K H KIM F04701-77-C-0120

UNCLASSIFIED SAMSO-TR-78-142 NL

1 OF 2

AD A064489

SAMSO—TR—78—142 LEVEL

METHODOLOGIES AND TOOLS FOR DEVELOPING ROBUST FTSC SOFTWARE

UNIVERSITY OF SOUTHERN CALIFORNIA
LOS ANGELES, CA 90007

31 AUGUST 1978

FINAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

PREPARED FOR

SPACE AND MISSILE SYSTEMS ORGANIZATION
AIR FORCE SYSTEMS COMMAND
LOS ANGELES AIR FORCE STATION
BOX 92960, WPC
LOS ANGELES, CALIFORNIA 90009

This final report was submitted by the University of Southern California, Los Angeles, California, under contract F04701-77-C-0120 with the Space and Missile Systems Organization, Los Angeles Air Force Station, Los Angeles, California. Second Lieutenant Douglas D. Orville (YCD) was the Project Officer.

Douglas D. Orville   Ronald G. Spray

DOUGLAS D. ORVILLE, 2LT, USAF
Project Officer

RONALD G. SPRAY, Lt Col, USAF
Deputy Director for Subsystems

FOR THE COMMANDER

[illegible]

LEONARD E. BALTZELL, Col, USAF
Asst. Deputy for Advanced
Space Programs

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. ~~REPORT NUMBER~~ (18) SAMSO	2. ~~GOVT ACCESSION NO.~~ (19) TR-78-142 (9)	~~3.~~ RECIPIENT'S CATALOG NUMBER
4. ~~TITLE (and Subtitle)~~ (6) METHODOLOGIES AND TOOLS FOR DEVELOPING ROBUST FTSC SOFTWARE		5. ~~TYPE OF REPORT & PERIOD COVERED~~ FINAL REPORT 1 July 77—31 August 78
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) K.H. KIM		8. CONTRACT OR GRANT NUMBER(s) (15) F04701-77-C-0120
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Southern California ~~Departments of Elec. Eng. and Computer Science~~ Los Angeles, CA 90007		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (16) 63401F, 2181 02 (17)
11. CONTROLLING OFFICE NAME AND ADDRESS Space and Missile Systems Org. (YCD) LAAFS, P.O. Box 92960, Worldway Postal Center Los Angeles, California 90009		12. ~~REPORT DATE~~ (11) 31 August 1978
		13. NUMBER OF PAGES 190 (12) 191 p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Fault-Tolerant Recovery Programs	Distributed Systems
System Recovery Strategies	Rollback Point Insertion
Error Recovery	PASCAL Implementation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is a report of the results that have been obtained from a research project aimed at learning methodologies and developing tools useful for obtaining reliable software of the Fault-Tolerant Spaceborne Computer (FTSC). Results are divided in three areas. First, a study was performed on the methods of designing well-structured recovery programs which are invoked on detection of an error to recover an operational system configuration and a consistent computation state. This study involved the experimental development of an FTSC recovery

361 550 ~~pg~~

program. This report discusses several program design and system recovery strategies that have been found useful in obtaining an easily understandable recovery program, together with the program developed. Second, a language processor was developed to facilitate experimenting with recovery block which is a language construct designed to support structured incorporation of program redundancy. It translates programs written in PASCAL augmented with recovery block into equivalent programs in ordinary PASCAL. The translation strategy used and the organization of the translator are described. Third, a new approach to error recovery in distributed systems of cooperating parallel processes was developed. In such systems processes must cooperate in recovery, in contrast to the previously studied approaches that require the program designer to coordinate the recovery point specifications of processes. the new approach relieves the programmer of that burden by using an intelligent processor system. Methods for efficient implementation of such a processor system were also developed.

METHODOLOGIES AND TOOLS FOR DEVELOPING ROBUST FTSC SOFTWARE

K. H. Kim

Departments of Electrical Engineering
and Computer Science
University of Southern California
Los Angeles, California 90007

August 31, 1978

Final Report for Period July 1, 1977 - August 31, 1978

Contract No: F04701-77-C-0120

Prepared for:

USAF (AFSC)
HQ Space and Missile Systems Organization
P. O. Box 92960
Worldway Postal Center
Los Angeles, CA 90009

Table of Contents

		Page
Abstract		i
Introduction		1
Part I:	Structured Design of a Fault-Tolerant Recovery Program of the FTSC	3
Part II:	A Translator of PASCAL Augmented with Recovery Block	76
Part III:	Programmer-Transparent Coordination of Recovering Parallel Processes	124
Acknowledgement		184

Abstract: This is a report of the results that have been obtained from a research project aimed at learning methodologies and developing tools useful for obtaining reliable software of the Fault-Tolerant Spaceborne Computer (FTSC). Results are divided in three areas. First, a study was performed on the methods of designing well-structured recovery programs which are invoked on detection of an error to recover an operational system configuration and a consistent computation state. This study involved the experimental development of an FTSC recovery program. This report discusses several program design and system recovery strategies that have been found useful in obtaining an easily understandable recovery program, together with the program developed. Second, a language processor was developed to facilitate experimenting with recovery block which is a language construct designed to support structured incorporation of program redundancy. It translates programs written in PASCAL augmented with recovery block into equivalent programs in ordinary PASCAL. The translation strategy used and the organization of the translator are described. Third, a new approach to error recovery in distributed systems of cooperating parallel processes was developed. In such systems processes must cooperate in recovery. In contrast to the previously studied approaches that require the program designer to coordinate the recovery point specifications of processes, the new approach relieves the programmer of that burden by using an intelligent processor system. Methods for efficient implementation of such a processor system were also developed.

Introduction

This is a report of the results that have been obtained from a research project carried out at University of Southern California under the sponsorship of U.S. Air Force - SAMSO during July 1, 1977 - August 31, 1978. The project aimed at learning methodologies and developing tools useful for obtaining reliable software of the FTSC. More specifically, research efforts were directed in three directions.

(1) to study the effective ways of designing well-structured and fault-tolerant recovery programs. (Recovery programs are the programs which are ~~re~~invoked, on detection of an error, to recover an operational system configuration and resurrect the interrupted computation.) Also to study the effective ways of using design redundancy to obtain fault-tolerant programs (i.e., programs capable of tolerating some residual design errors in them).

(2) to develop a language processor to facilitate structured fault-tolerant programming.

(3) to study the structure of fault-tolerant distributed systems.

The results are described in three parts, following this section.

Part I describes some program design and system recovery strategies that have been found useful in obtaining an easily understandable recovery program of the FTSC. The strategies are of such general nature that they can be useful in designing recovery programs of other modular fault-tolerant computers. It also presents a PASCAL specification of a FTSC recovery program that has resulted from the application of the strategies.

Part II documents a language processor that translates programs written in PASCAL augmented with recovery block into equivalent programs in ordinary PASCAL. The recovery block facilitates structured incorporation of program redundancy. The basic translation strategy is discussed and then the full listing of the translator is given together with some test-run results.

Part III discusses a new approach to error recovery in distributed systems of cooperating parallel processes. More specifically when each process is capable of error detection, rollback, and retry, the recovery points of the processes must be properly coordinated to prevent a disastrous avalanche of process rollbacks. In contrast to the previously studied approaches that require the program designer to coordinate the recovery point specifications of processes, an approach of relieving the programmer of that burden was developed. The new approach relies upon an intelligent processor system (that runs processes). Basic rules of reducing storage and time overhead in such a processor system were also developed.

Part I

STRUCTURED DESIGN OF A FAULT-TOLERANT RECOVERY PROGRAM OF THE FTSC

by

K. H. Kim, J. Huang, and M. Naghibzadeh

Abstract: This paper reports the results obtained in an experimental project aimed at developing a well-structured and effective recovery program of the FTSC (Fault-Tolerant Spaceborne Computer). This experiment was also chosen as a means of studying the ways of using design/program redundancy to improve the software reliability, mainly because of the relatively small program size and the unusual complexity in analyzing the situations that the recovery program must deal with. The paper presents several program design and system recovery strategies that have been found to be useful in obtaining an easily understandable recovery program of the FTSC but may also be useful in development of recovery programs of other computers. It then discusses a recovery program for the FTSC that has resulted from the application of the strategies. The recovery program has been specified in PASCAL and its conversion into an assembly language program is briefly discussed. The listing of the PASCAL specification is given in an appendix.

1. Introduction

This is a report of the results obtained in a research project that involved the development of a program of the FTSC (Fault-Tolerant Spaceborne Computer) [B1,S1,S2] responsible for recovering an operational system configuration and resurrecting the interrupted computation. The project was launched with two major objectives:

(1) to identify good structures and design strategies for the software that controls recovery from hardware faults, and

(2) to find effective ways of using design/program redundancy in development and maintenance of robust real-time software;

First, the systems employed in critical applications requiring ultra-reliability commonly contain large amounts of redundant hardware. The redundant hardware is provided primarily to facilitate fault detection and recovery. Recovery, following the fault detection, is typically a function of both hardware and a special software called recovery program. The hardware automatically recovers certain modules, collectively called a hardcore, needed for correctly running a recovery program, and then the recovery program recovers additional hardware modules, establishes an operational configuration, and recovers the information needed to resurrect the interrupted computation. The performance of a recovery program is crucial to the success of the entire application requiring ultra-reliability. A recovery program must be both efficient and reliable itself. Also during the design of a recovery program, considerations must be given to the possible faults of the hardware modules on which the recovery program runs, i.e., the CPU and the memory containing the recovery program. The redundant hardware performing fault detection may also be faulty. Engineering an efficient and robust recovery program is thus unusually complex. In spite of the important role that a recovery program plays in any fault-tolerant system, very little has been published in this field and many fundamental questions concerning good structures and design strategies of a recovery program have yet to be answered.

Secondly, the study of the methods of using program redundancy, frequently called fault-tolerant programming, was motivated by its potential as a means of tolerating residual design errors/inadequacies [H1,H2,K1,R1]. In spite of recent advances in rigorous specification, structured design, program validation, and development of modern high level languages, large-scale software is still put into operation with residual design errors/inadequacies. Thus fault-tolerant programming appears to be an attractive supplement to those already widely accepted practices for the development of real-time software required to be ultra-reliabe. In particular, a language construct, called recovery block, developed by Horning et al [H3,R1] supports the incorporation of program redundancy in a well-structured form. It thus paved a way to extensive use of program redundancy without degrading the program readability and provided a further stimulus to the initiation of this study. Appendix A briefly summarizes the syntax and semantics of the recovery block.

As a means of identifying problem areas and testing various potential solutions, the design of a recovery program of the FTSC was chosen. A recovery program of the FTSC was considered to be an ideal subject of a fault-tolerant programming experiment (in addition to being an obvious subject of a recovery program engineering experiment), due to the relatively small program size and the unusual complexity of the situations that the recovery program must deal with. In addition, the FTSC is a modular computer equipped with a powerful set of hardware features supporting fault detection and recovery [S1,S2]. This rich set of fault-tolerant hardware features offers a number of options in designing a recovery program, thereby providing ample opportunities for employing recovery blocks with benefits. That is, the primary recovery procedure, alternate recovery procedures, and the acceptance test can use independent test and configuration techniques. Therefore, although a conventionally structured recovery program has operated satisfactorily on the brassboard FTSC, an experimental development has been undertaken under the sponsorship of the U.S. Air Force - SAMSO to study the application of fault-tolerant programming techniques to such a program. The recovery block provides a means of incorporating multiple recovery procedures in a well-structured form, but the individual procedures must also be systematically designed and easy to understand to increase the chance of obtaining a robust recovery program. This requirement, although not furthur discussed here, was also implemented in the resulting program.

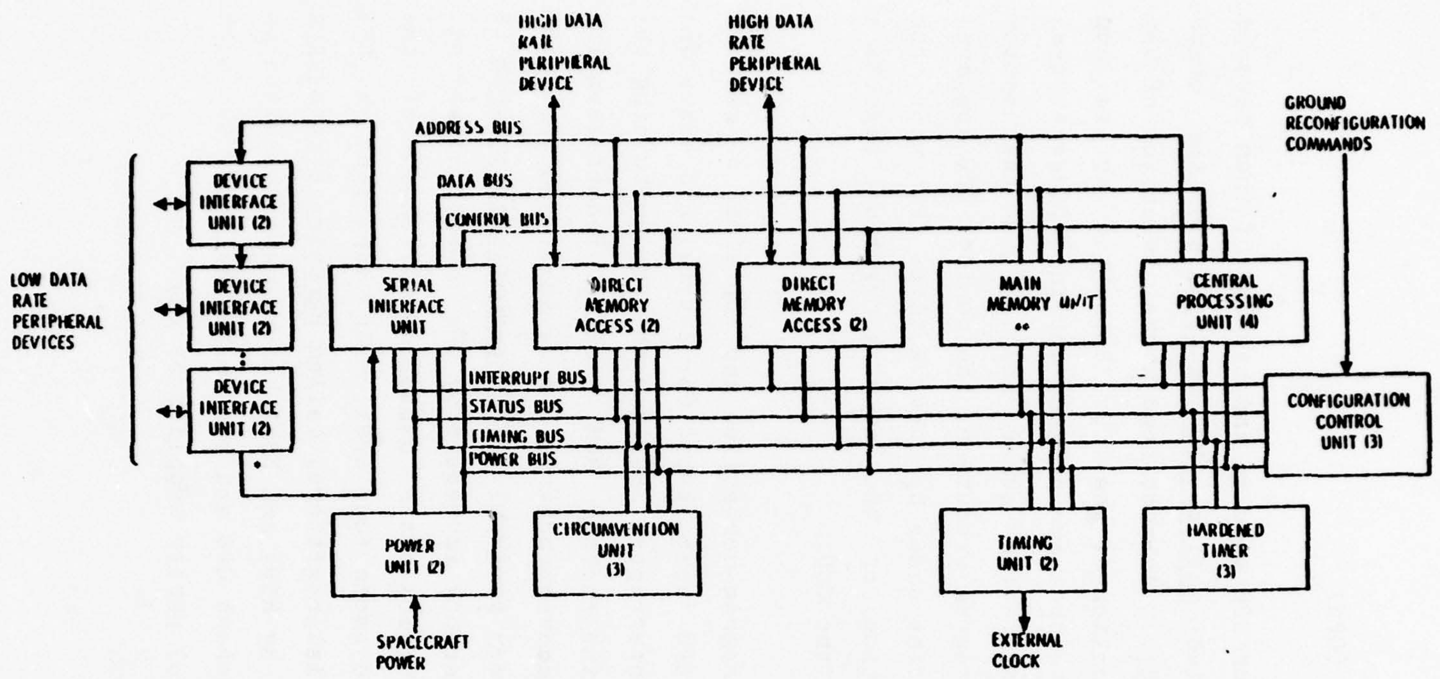
Although the FTSC has been documented fully in [F1,S1,S2], we have included a sketch of its features to make this report self-contained. Section 3 discusses the adopted organization of the design process. Several basic design strategies that have been found to be useful in obtaining an easily understandable recovery program are discussed in section 4. Section 5 describes a model of the FTSC devised to support the development of a systematic recovery procedure. The procedure obtained is discussed in section 6. Section 7 provides a brief introduction to the PASCAL specification and section 8 briefly discusses the aspect of converting the PASCAL specification into an assembly language program. Appendix B contains a complete recovery program specified in PASCAL (augmented with recovery blocks). Appendix C supplements the PASCAL specification in Appendix B with more details on some subprocedures of testing modules.

2. A sketch of the FTSC structure

The FTSC is a 32-bit microprogrammable machine designed to operate in long-duration space missions through radiation and spacecraft discharge events. It is intended to provide a five-year on-orbit capability with 95% probability of survival [B1,S1]. Figure 1 depicts the configuration of the FTSC. Every functional unit (shown in Figure 1) contains a certain amount of hardware redundancy. Each unit is briefly described below and many features irrelevant to fault detection and recovery are not described.

2.1 Configuration control unit (CCU)

The FTSC contains more CPU modules and lines on the address and data buses than required by an operational system configuration. Selection of a subset of those components to form an operational (CPU - bus) configuration is a function of the CCU. In addition, the CCU receives reports from the fault detectors, also called monitors, distributed throughout various functional units (shown in Figure 1). For example, each module that uses the address bus (A-bus) and/or data bus (D-bus) contains a bus code monitor which detects invalid codes on the bus. When a fault is detected by one or more functional units (more precisely the monitors contained in them), the condition is reported to the CCU. Upon receiving fault reports the CCU categorizes the faulty situation, establishes a new CPU-bus configuration if the performance of the present configuration is in doubt, and then forces the CPU to execute the recovery program through an interrupt called fault interrupt. Throughout the execution of the recovery program the categorized information on the detected fault is supplied to the CPU. The fault categorization scheme used by the CCU is described at the end of this section. The CCU is clearly a hardcore unit of the FTSC and is TMR- (triple modular redundancy) configured.


HIGH DATA RATE PERIPHERAL DEVICE
HIGH DATA RATE PERIPHERAL DEVICE
GROUND RECONFIGURATION COMMANDS
ADDRESS BUS
DATA BUS
CONTROL BUS
LOW DATA RATE PERIPHERAL DEVICES
DEVICE INTERFACE UNIT (2)
DEVICE INTERFACE UNIT (2)
DEVICE INTERFACE UNIT (2)
SERIAL INTERFACE UNIT
DIRECT MEMORY ACCESS (2)
DIRECT MEMORY ACCESS (2)
MAIN MEMORY UNIT **
CENTRAL PROCESSING UNIT (4)
CONFIGURATION CONTROL UNIT (3)
INTERRUPT BUS
STATUS BUS
TIMING BUS
POWER BUS
POWER UNIT (2)
CIRCUMVENTION UNIT (3)
TIMING UNIT (2)
HARDENED TIMER (3)
SPACECRAFT POWER
EXTERNAL CLOCK

* Up to 60 device interface units (DIUs)
( ) Indicates number of modules, including spares

** User option, up to 15 active and 9 spare modules. The typical use requires 4 active modules and 3 spares (for 5 years, 0.95 computer reliability)

Figure 1. FTSC Block Diagram

2.2 Central processing unit (CPU)

The system contains four CPU modules and has two of them powered on at any one time, one called active CPU (A-CPU) and the other called monitor CPU (M-CPU). As mentioned above, selection of two powered CPU modules is a function of the CCU. Both the active and monitor CPUs always execute the same program but in different ways. While the active CPU interacts with other modules in the system through the buses during program execution, the monitor CPU compares the information loaded onto the buses by the active CPU with the result of its own execution of the same program and reports a disagreement, if occurs, to the CCU.

A CPU module is microprogram-controlled and contains 8 general purpose registers. The CPU uses 16-bit (or 2-byte) addresses and acknowledges 10 levels of interrupts. Besides the ROM containing the microprogram, the CPU contains another ROM, called reconfiguration ROM, which contains a recovery program (to be exact, an initially executed segment of a recovery program). This reconfiguration ROM is a CPU-resident part of the memory addressed by the 16-bit addresses. Therefore, the system contains four copies of (a segment of) the recovery program distributed among four CPUs. In addition, a CPU contains a pair of special registers, called hardware status word (HSW) registers and denoted by HSW1 and HSW2, carrying information on system status (i.e., the current CPU and bus configurations, the most recently reported fault, etc) partly supplied by the CCU.

The instruction set of the FTSC supports 32-bit fixed-point and floating-point arithmetic as well as vector operations that are particularly useful for navigation and pointing applications. Direct and indirect addressing with predecrement and post increment are available. Some instructions are specifically oriented for testing the fault-tolerant features. For example, data words can be stored with bad parity in order to exercise error-detection and correction features in the main memory unit (MMU) modules. Some instructions are designed to be executed differently by the A-CPU than by the M-CPU. Such instructions can thus be used to cause disagreement between the two operational CPUs.

There are instructions for powering on/off the DMA, SIU, and MMU modules. There are also instructions, which may be called CCU commands (these are called "program flags" in [S1]), by which the CPU can cause the CCU to change its state and thereby change the system configuration (including the selection of CPU modules and bus lines to be used).

2.3 Main memory unit (MMU)

The MMU can consist of up to 24 non-volatile and non-destructive readout memory modules, each containing 4K words. As many as 15 modules can be powered on at any one time, with each identified by a changeable, but unique, four-bit soft (module) name. A properly functioning memory module responds to all addresses whose four most significant bits correspond to the current soft name of the module. On the other hand, each module can be accessed by a permanent and unique module number, called hard name, for the purpose of powering on/off, assigning a new soft name, etc.

An MMU module contains 41 bit lines of which 32 are used to contain data information, 6 to carry parity codes for error detection and correction, and 3 as spare bit lines. Therefore, up to three bit lines may be lost without disabling the module and the information contained in a module can be recovered and saved into another module as long as the number of lost bit lines does not exceed four.

2.4 Two direct memory access (DMA) units

Two DMA functional units DMA1 and DMA2, are provided. Each is redundant, consisting of a pair of modules of which one is powered on at any one time. Each DMA function provides a direct parallel access between an external user (i.e., peripheral equipment) and the MMU.

2.5 Serial interface unit (SIU)

The dual redundant SIU functions much the same way as the DMA *but provides serial access for up to sixty users.* Only one SIU module is normally powered on at any one time.

2.6 Power unit (PU)

The power supply of the FTSC is provided by the redundant PU that consists of two modules, each containing dual <u>output (voltage) monitors</u>. If either monitor finds a (voltage) out-of-tolerance condition, it immediately announces this condition to the rest of the system and effects a switchover to the alternate module. Therefore, the PU is an automatically recovering unit.

2.7 Timing unit (TU)

The TU supplies clock pulses to the rest of the system and consists of two modules, each containing dual output monitors. As with the power monitors, either monitor can cause a switchover to the alternate module. Again, the TU is an automatically recovering unit.

2.8 Circumvention unit (CU)

This TMR-configured unit detects radiation events and (upon detection) clamps other units in the system until the radiation disappears, thereby protecting them from possible damage.

2.9 Hardened timer (HT)

The HT is used to count the amount of elapsed time during the periods of PU failure or high radiation event. It is TMR-configured.

2.10 Bus network

There are seven buses. The A-bus (carrying 16-bit addresses) and the D-bus (carrying 32-bit data words) use cyclic error-correcting codes with 8 parity bits (i.e., 1 parity code byte) and contain one spare byte each. Thus the A-bus consists of four bytes of lines and the D-bus consists of six bytes of lines. As mentioned earlier, selection of the bytes to be used is performed by the CCU, and the CPU can also cause the CCU to reconfigure the A-/D-bus by executing a CCU command. In addition, there are the control bus (C-bus) used in conjunction with the A- and D- buses, the interrupt bus (I-bus) carrying interrupt signals, the status bus (S-bus) carrying fault reports and reconfiguration signals, etc.

2.11 Additional details on the CCU

Each time a CCU command for reconfiguring the CPU, the A-bus, or the D-bus is executed, the new subset of components is chosen such that the same subset is not repeated until all subsets have been tried. There are a number of different states, where the CCU establish different system configurations, that the CCU can be in. To be more specific, the CCU can be viewed as containing five flip-flops:

the first called <u>reconfiguration state flag</u>,
the second called <u>alexic state flag</u>,
the third called <u>flag 1</u>,
the fourth called <u>flag 2</u>, and
the fifth called <u>flag 3</u>.

There is a CCU command by which the CPU can clear all five flags at once. These flags are used as follows.

(1) The reconfiguration flag is set (i.e., the CCU enters a reconfiguration state) when a fault interrupt is generated by the CCU (after receiving a fault report in a normal state).

(2) On receiving a report of a PU failure or high radiation event, the CCU enters an alexic state (by setting the alexic flag) and this state transition causes a signal to be generated for turning off the DMA, SIU, and MMU modules.

(3) Flag 1, flag 2, and flag 3 can be set only by execution of a CCU command (by the CPU). When flag 1 is not set while the CCU is in a reconfiguration state, the DMA, SIU, and MMU modules shall not respond to any address on the A-bus. By setting flag 1 the recovery program can enable the modules to respond to their addresses and will do this at its convenience. In fact, flag 1 was provided primarily to enable CPU and bus tests without affecting the DMA, SIU, and MMU modules. Thus flag 1 can be used to indicate that the CPU has been validated. Flag 1 can be reset in three different ways: PU failure or high radiation event, CPU reconfiguration due to detection of a CPU fault, or execution of a CCU command.

(4) Flag 2, when set, inhibits CPU reconfiguration and generation of fault interrupts due to various faults except PU failure, high radiation events, and TU reconfiguration (i.e., automatic module switchover). Flag 2 is reset on detection of <u>any</u> fault and can also be reset by execution of a CCU command.

(5) Flag 3, when set, inhibits generation of fault interrupts caused by the reports implicating the A-bus, D-bus, DMAs, SIU, or MMU. Flag 3 is reset on detection of <u>any</u> fault and can also be reset by execution of a CCU command.

The fault categorization scheme used in the CCU is the following.

Cat I: PU failure or detection of radioactive events by the CU,
Cat II: CPU implicated (e.g., disagreement between A- and M- CPUs),
Cat IIIa: A-bus implicated (e.g., detection of an invalid code on the A-bus by more than one bus code monitors),
Cat IIIb: D-bus implicated,
Cat IV-DMA1: DMA1 implicated (i.e., the DMA1 is the only unit that detected an invalid code on one of the buses),

IV-DMA2: DMA2 implicated,

IV-SIU: SIU implicated,

IV-overrun: An MMU, DMA, or SIU module has not responded to a request within the predetermined period,

IV-TU: TU reconfiguration (i.e., switchover) has occurred.

While the system is in a reconfiguration state, the CCU supplies the category information about the most recently reported fault to the CPU (to be exact, to a field of the HSW1).

3. Organization of the design process

At the beginning of this project a plan was made to carry out the design study through four major steps:

(1) modeling of the FTSC,

(2) identification of feasible recovery procedures,

(3) specification of a recovery program in PASCAL, and

(4) study the aspect of coding a recovery program in the FTSC assembly language.

The rationale for this plan is given below.

First, it was felt that systematic search for various effective recovery procedures could be greatly assisted by working with a machine model which hides many recovery-irrelevant features of the sophisticated FTSC. Modeling of the FTSC was also motivated by the hope that a properly chosen model would facilitate the validation of the capability and efficiency of a resulting recovery program.

Secondly, after determining a recovery procedure based on a FTSC model, one could directly convert the procedure into a final recovery program (i.e., an assembly/machine language program). However, it seemed clear that validation of such a recovery program would be inefficient, especially considering the numerous unusual situations that a recovery program must circumvent and the large number of test cases that may be required. Modification or debugging of such a program would also be difficult. Therefore, the chance of success in obtaining a robust recovery program was expected to be greatly enhanced by obtaining a high level language version of a recovery program first, checking it out by a compiler, and then developing a final recovery program by using the high level language version as a specification. The language chosen is PASCAL. The natural control primitives (e.g., if-then-else, case, while-do, etc) would relieve the designer of the burden of implementing equivalent control sequences in inconvenient assembly language primitives. In fact, it was necessary to augment PASCAL with a recovery block to experiment fault-tolerant programming. A translator that translates a program written in PASCAL augmented with the recovery block into an equivalent program in ordinary PASCAL, has been developed. It is documented in [K2].

4. Design strategies

Several strategies that have been followed throughout the design and found to be useful in obtaining an easily understandable recovery program, are now discussed.

4.1 Fixed ordering of units for recovery

When fault monitors are distributed among various modules, it is highly tempting to check and recover modules in varying orders depending upon how some modules indict the others. However, this is a dangerous strategy, considering the possible occurrence of multiple faults (i.e., simultaneous occurrences of faults in multiple modules or failure of a module before another already faulty module is located and amputated) as well as the indictment of operational modules made by the faulty modules. In other words, the behavior of a recovery program based on such a strategy in the presence of multiple faults is difficult to predict.

For example, assume that from the received fault reports the CCU has identifed a certain fault category that may be caused by either a malfunctioning CPU or a malfunctioning (A-/D-) bus. If the CPU is first tested using the untested bus and the actual faulty unit is the bus, disagreement between the A-CPU and the M-CPU may result and lead the recovery program to implicate the CPU of being faulty. On the other hand, if the bus is first tested using the untested CPU while the CPU is malfunctioning, the CPU could mistakenly cause a good bus to be diagnosed as the faulty one. Worse yet, bus reconfiguration and reentry of the malfunctioning CPU into the recovery program could lead to an endless loop. In this sense an endless loop means an infinite repetition of entering the recovery program caused by an unidentified faulty module while there remain a sufficient set of operational modules to form an operational configuration.

To alleviate such problems, it is useful to observe the rule of <u>recovering units in the same fixed order and using only the already validated (and recovered) units in recovering other units</u>. Under the rule the total recovery process becomes a <u>strictly linear sequence of unit recoveries</u>. To illustrate some implementation aspects of the rules, consider a system of three units ordered as shown in Figure 2. Unit U1 occupies the first position and is assumed to be hardcore. Assume that U1 is capable of performing the function of the CCU in the FTSC, i.e., receiving fault reports and invocation of the recovery program, and also that the execution of a recovery program can be started by using U1 only. The initial segment of the recovery program first tests and recovers unit U2 and during this process it must not use unit U3 in any way. Furthermore, if unit U3 contains a monitor which can indict unit U2 and can cause the recovery program to be reentered, then either the monitor must be disabled or U1 must be armed to ignore a fault report from the monitor (during the testing of U2) since U3 cannot be trusted yet.

After recovering U2, the recovery program proceeds to test and recover U3 and during this process the recovery program may rely upon the monitoring capabilities of U1 and U2. At this time it is possible that U2 is faulty because of two reasons although both cases have very low probability of occurrence. First, even though U2 was recovered earlier, it could have become faulty by the time when U3 is under test. Second, the recovery program could have misjudged a faulty U2 module as an operational module and (thus misjudged that U2 was recovered). Therefore, the recovery program must be designed such that when all possible configurations of U3 have been tried and have failed in validation, the configuration of U2 is changed and validation of U3 is reattempted. It is of course possible that the monitors in U1 and/or U2 detect the faulty condition of U2 and report to U1 before all possible configurations of U3 are tried, in which

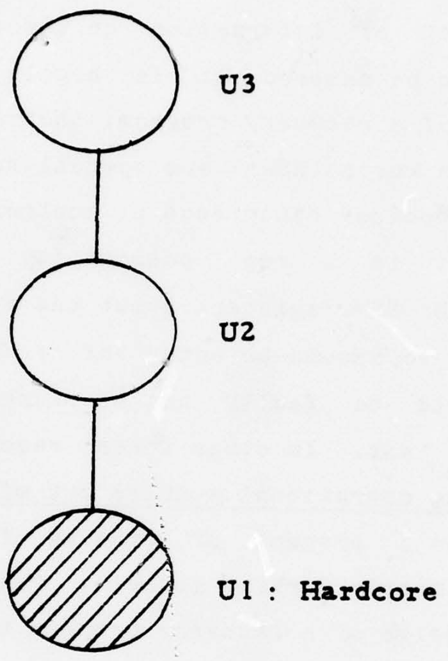


Figure 2 A system consisting of three functional units

case U1 will reconfigure U2 and reinvoke the recovery program.

4.2 No trust of a module by default

Since the FTSC hardware (especially the CCU) keeps a considerable amount of information on the fault that has caused a recovery program to be entered, it is highly tempting to assume, during the design of a recovery program, that units not implicated by the hardware status words (HSWs) are operational. Such an assumption may not be valid because occurrence of another fault during recovery of a previous fault is a real possibility and may destroy the information in the HSW registers about the first fault. Therefore, it was made a rule to assume on entry of a recovery program that every module could be faulty and to trust a module only after it passes an explicit test. In other words, recovery was regarded as a <u>process of finding operational modules out of all potentially faulty modules</u> rather than a process of finding faulty modules from a configuration of mostly working modules. This conservative approach facilitates the design of a recovery program that avoids entering an endless loop.

4.3 Redundant design

Unlike the behavior of a correct machine, the behavior of a faulty machine is very difficult to predict. From the beginning it was expected to be a difficult task to find a single recovery procedure which works under all possible circumstances. Therefore, it was also made a rule to mobilize a set of alternate procedures. These alternates can be specified in a well-structured form by use of the recovery block. Additional specific justifications for this redundant design are given in section 6.

4.4 Simplicity over efficiency

Finally it was decided that wherever there is a trade-off between simplicity and efficiency, simplicity would be given a priority. Only if a recovery program resulted from this strategy fails to meet the performance requirement, then various ways of optimizing the program, preferrably without changing the overall program structure, can be sought for.

5. Basic functions of a recovery program and modeling of the FTSC

As mentioned before, a recovery program in the FTSC is responsible for establishing an operational hardware configuration and recovering some of the lost information in the memory. In other words, a recovery program identifies a set of operational modules sufficient for normal processing and then restores the system to an executable state (i.e., restores important memory contents and conditions peripheral units to the states ready for normal processing by appropriately setting control registers contained in them). (On completion of the execution of a recovery program, the executive (i.e., supervisor) program will enter and schedule application tasks to restart from their rollback points.) A useful FTSC model must therefore illuminate the procedure of expanding the set of trusted modules until an operational configuration is established. A natural modeling criteria is the operational precedence among functional units (each consisting of multiple modules). If unit A has higher operational precedence than unit B, then unit B cannot operate reliably unless unit A is operational. Figure 3a depicts a FTSC model displaying the operational precedence among units. The rationale for the ordering in Figure 3a is the following.

(1) Since the PU supplies the power to all other units, it must be working reliably before any other unit can be used. Naturally it has the highest operational precedence.

(2) The unit having the next highest operational precedence is somewhat arbitrarily chosen to be the CU. It depends only on the PU for proper functioning.

(3) The operation of all other units (excluding the PU and the CU) depends upon the correct clock signals supplied by the TU. Thus the TU has the third highest operational precedence.

(4) The HT maintains the record of elapsed time during the periods of PU failure and high radiation event. The proper operation of the HT is dependent only upon the PU, CU, and TU. The CCU is also dependent

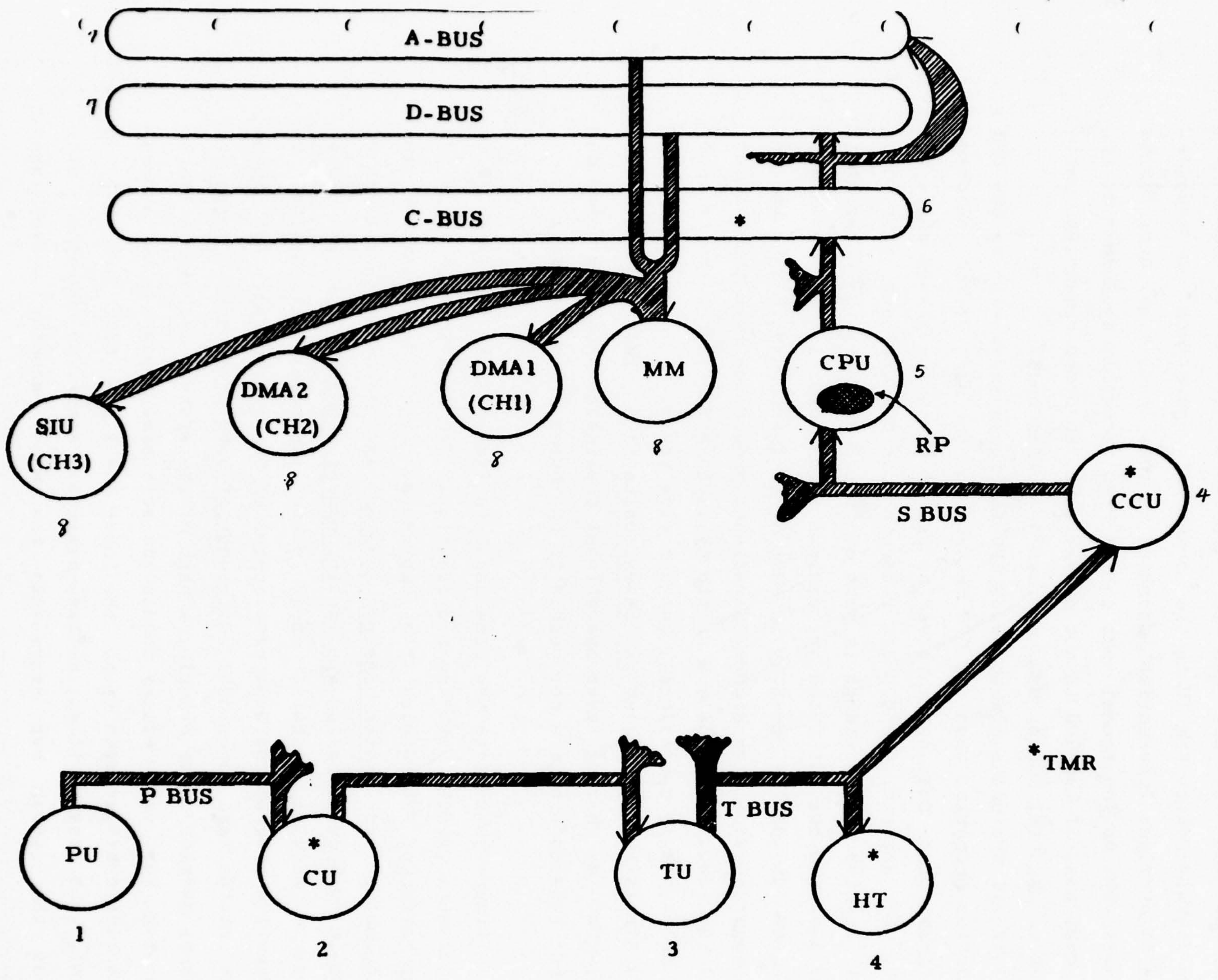


Figure 3a Operational Precedence among the functional units in the FTSC

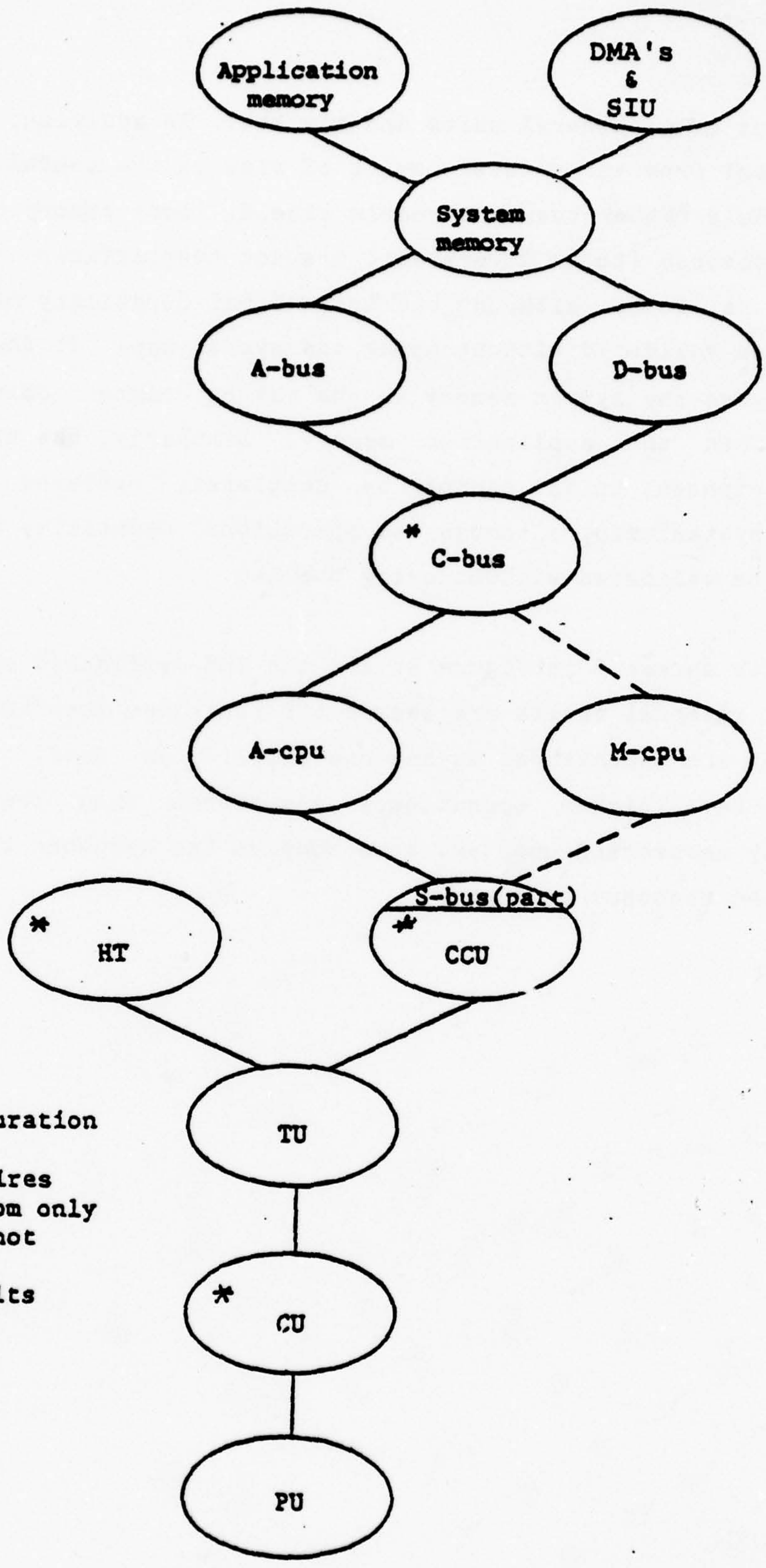
only on the PU, CU, and TU. Therefore, both the HT and the CCU immediately follow the TU in the order of operational precedence. The HT provides information important to some (real-time) application tasks, but no functional unit in the FTSC is really dependent on it. In contrast, if the CCU is not operational, no other remaining units such as the CPU, buses, MMU, etc can be relied upon.

(5) If all the units placed below CPU in Figure 3a as well as the CPU function properly, then the CPU can execute the part of the recovery program that does not involve the use of the units placed above the CPU. For example, the CPU can interact with the CCU and this does not involve buses except (a part of) the S-bus. On the other hand, the A-bus, D-bus, or C-bus are useless if the CPU is not available because the CPU is normally in control of these buses. Thus the CPU is regarded as having higher operational precedence than those buses.

(6) The A-bus and D-bus are usable by various functional units only if the C-bus is operational, but not vice versa.

(7) Similarly the MMU and peripheral units (i.e., DMAs and SIU) are regarded as having lower operational precedences than the A-bus and D-bus because they are not usable if the buses are not working.

Figure 3b depicts the same model in Figure 3a in the form of a precedence graph with one modification. The modification is in the separation of MMU modules into two sets: one consisting of two modules called system memory modules and the other consisting of remaining modules called application memory modules. The two system memory modules contain, among other important information, an executive program and duplicate copies of the system map that shows the status and functional assignment of both peripheral modules and memory modules. For example, a part of the system map is a table showing for each memory module the soft name assigned, the current bit line configuration (i.e., the three bit lines that are not in use), the status (i.e., active, available spare, or unusable), etc. Thus the system map represents the most recently established



★ : TMR configuration

※ (faults of) wires coming out from only one unit are not distinguished from the (faults of the) unit.

Figure 3b A precedence graph model of the FTSC

configurations of peripheral units and the MMU. In addition, what is more important from the recovery point of view is the useful content of an MMU module rather than the module itself. Some memory contents cannot be recovered (to be more exact, are not identifiable) if the system map is lost, although the operational capability of memory modules can be validated without using the system map. It thus seems useful to regard the system memory as one having higher operational precedence than the application memory. Similarly, the execution states of peripheral units cannot be completely restored without using the system map, although the operational capability of their modules can be validated without using the map.

The units marked * in Figure 3b are the TMR-configured ones and thus their internal faults are masked off (privided that two out of three modules are operational at any one time). In fact, all the units that have higher operational precedence than the CPU are automatically recovering ones and thus compose the hardcore that does not burden the recovery program.

6. Recovery procedure

In principle, the overall recovery process can be designed in a straightforward manner on the basis of the FTSC model presented in the preceding section. The process should essentially be a step-by-step expansion of the set of trusted modules in a decreasing order of the operational precedence. The order in which the modules having the same operational precedence are tested and recovered is immaterial as long as the same fixed order is used consistently.

In the FTSC, TMR-configured and automatically recovering units form the hardcore. No capability is provided within the FTSC for direct repair or reconfiguration of the hardcore. (The FTSC was designed, however, to allow the ground station to intervene and control various functional units after failure of more than one CCU module. This ground-override mode is outside the scope of this paper.) Thus the correct operation of these functional units must be trusted by a recovery program without explicit checks.

When a recovery program is entered, a certain amount of information on system status including the categorized fault report is available from the HSWs. By utilizing this information a recovery program can establish a working configuration in an efficient manner. Basically faults of multiple modules causing an invocation of a recovery program are less probable than faults in a single module. In other words, if the categorized fault report implicates a certain module of being faulty, then the probability of other modules being faulty at the same time is small. Thus, the recovery program can check the implicated module more thoroughly than other modules. This strategy of using "weak" and "strong (thorough)" tests is of course somewhat against the aforementioned rule of not trusting any module by default. However, it is a way of reducing the (average) execution time of the recovery program, thereby reducing the probability of the program failing in meeting the execution time requirement. Later an analysis may reveal that the worst-case execution time of a final recovery program is far less than the tolerable limit. In such case, some weak (module) tests can be deleted to leave only strong tests in the recovery program, thereby increasing the robustness of the program even further. A penalty incurred by using a~~n~~ weak test is the increased probability of a faulty module being undetected (i.e., validated). Therefore, in order to make use of weak tests without much sacrificing the capability of correct recovery, an acceptance test needs to be provided at the end of a recovery program to ensure that the entire computer system has been correctly recovered. An alternate procedure which uses only strong tests must also be provided to recover the system if the result of the primary procedure is rejected by the acceptance test. Thus the use of weak and strong tests is to trade the increase of the worst-case recovery time for reduction of the average recovery time.

The transient faults are expected to be the predominant fault-type. The occurrence of a transient fault can be determined if the module that has been indicted passes a (strong) test. Nevertheless, the source of the transient fault is not always easy to locate. For instance, if the A-bus has been indicted but both the A-bus and its most recent user DMA1 (that loaded an address on the bus at the time of fault reporting) pass (strong) tests, then one may conclude that a transient fault has occurred. Both the A-bus and DMA1 will be suspected as the possible sources of the transient fault but the exact source cannot be identified. It is useful to maintain a transient fault count and to set a tolerable limit to the count for each module. When this limit is reached, the associated module can be treated as an unusable module and thus replaced by a standby module. The procedure of updating transient fault counts is not detailed in this paper.

The recovery consists of the following sequence of actions.

(1) validation of the CPU's capability of reading the hardware status words,

(2) validation and necessary reconfiguration of the CPU,

(3) validation and necessary reconfiguration of the A-bus and the D-bus,

(4) recovery of the system memory,

(5) recovery of peripheral units,

(6) recovery of the application memory,

(7) acceptance test of the new system configuration (i.e., result of the recovery) and normal processing restart.

The rest of this section briefly explains the above actions.

The first action taken by a recovery program is to check if the present CPU can correctly read the HSWs. Failure of this validation will result in reconfiguring the CPU (i.e., selecting a new pair of CPU modules as the A- and M- CPUs) and reinvoking a recovery program. If the validation is successful, then the recovery program will proceed to check the CPU (to be exact, the part of the CPU not used in reading the HSWs) because the CPU has the highest precedence among the units that need to be checked.

If the HSWs indicate the occurrence of a CPU reconfiguration, then the (current) CPU is thoroughly tested. Otherwise, a weak test procedure is used. In any case, if the CPU fails in validation, it must be reconfigured. (Recall the CCU command provided for this programmed reconfiguration.) Both weak and strong test procedures for the CPU are detailed in Appendix C. By the rule of fixed-order unit recovery, a CPU test procedure must not use any unit (including the A- /D- bus) having a lower operational precedence than the CPU. This means that a faulty condition of the CPU must be detected by the CCU (that has the higher operational precedence than the CPU).

Therefore, a CPU test procedure is generally designed to compute a function by using certain parts of a CPU module and then send a CCU command (through a part of the TMR-configured S-bus) if the computation result is in a certain expected range. The function must be chosen such that if the parts of a CPU module under test are faulty, the probability of the result being out of the expected range and thus no CCU command being sent out is very high. When the CCU receives a CCU command from only one of the two powered CPU modules, the CCU reconfigures the CPU. Since the M-CPU monitors the C-bus and the C-bus is TMR-configured, a thought was given at first to the possibility of using the C-bus in validating the CPU although the C-bus is placed above the CPU in Figure 3b. This could not be implemented because the FTSC instruction set did not include an instruction which changes the C-bus only (without affecting the A-/D-bus).

Use of a weak test procedure can be justified only if a strong test procedure requires a large execution time and the weak test procedure can detect a faulty module with much reduced execution time and yet with little reduced effectiveness of diagnosis. This was not the case in validating the A-bus and the D-bus. Therefore, only a strong test procedure was provided. In addition, the same test procedure can be applied to both the A-bus and the D-bus.

By the rule of fixed-order unit recovery, the validation of the A-/D- bus must be performed free of the interference by the units having lower operational precedences than the A-/D- bus. The FTSC was designed to ease the implementation of this procedure. If flag 1 is not set while the system is in the reconfiguration state, the MMU, SIU, and DMA modules do not attend to the buses. Thus detection of a faulty bus relies upon the bus code monitors contained in the CPU. (It was assumed that the M-CPU will report to the CCU the disagreement with the A-CPU only if the information on the bus is a valid code.) The test procedure involves repeatedly placing a '0' and a '1' on each bit position of a bus.

Having established a~~n~~ working configuration of the CPU, the A-bus, and the D-bus, the next step is to recover the system memory modules. Flag 1 is first set so that the MMU, DMA, and SIU modules may respond. Now whenever a word is loaded onto a bus, a peripheral module containing a 'crazy' bus code monitor can report to the CCU the detection of an invalid code on the bus even though the bus is correct and the word loaded is a valid code. It is also possible that a malfunctioning (powered) peripheral module may respond to an address which is not the one assigned to itself. Since the system memory recovery must be performed free of interference by the units having lower operational precedences than the system memory, all the peripheral units are powered off before search for the system memory modules begins. During the execution of a power-off command for a peripheral unit, another peripheral unit may interfere by generating a Cat IV fault since the command involves the use of the A-bus. Therefore, flag 3 must be set in advance to mask such fault reports. Note also that a module may "resist" to obey the power-off command; it can then be powered off only when the system enters an alexic state.

A system memory module is assigned a soft name 0 or 1. An operational system memory module contains a special bit pattern, called a <u>map flag</u>, in its last location (address: 4095). Therefore, even if the contents of soft name registers in system memory modules are lost due to PU failure or other faults, it is possible to identify a system memory module by checking the last location in non-volatile memory. In the FTSC, a memory location can be read only by a soft address. It may seem reasonable that if it is suspected that most memory modules are powered on (after setting flag 1), then a memory read can be commanded with soft name 0 (or 1) and location address 4095 to see if the value read is a valid map flag. However, a malfunctioning memory module could have 0 as its soft name and thus more than one memory module may respond to soft name 0 (or 1). Therefore, it was decided to turn all the memory modules off and then power at most one of them on at any time during the search for system memory modules. (In the case where volatile memory modules are used together with backup batteries, the above strategy may be overly conservative and a different strategy may be desired.)

The search for system memory modules will result in one of three different situations: none found, only one found, and both system memory modules found. In the first (infrequent) case where no system memory modules have been found, a "cold restart" procedure is entered to establish the system memory by using the information stored in the backup memory. The cold restart involves finding two operational memory modules and one operational DMA module and then establishing the system memory (i.e., loading system software including the executive program from a backup memory into the two memory modules using the operational DMA module and initializing system maps in the modules). In the second case where only one system memory module has been found, the module is tested. If it passes the test, then another operational memory module (preferably a module designated as a spare in the system map contained in the just validated system memory module) is found and made the other system memory module. That is, the system map is copied into the newly assigned system memory module and some system software is loaded onto the module from the backup memory. If the first found system module fails the test, then the cold restart procedure is entered. In the third case when both system memory modules have been found, the two modules are tested and if any of them is found to be faulty, the procedures used in the previous two cases can be used again to recover the entire system memory.

With the properly recovered system map, an operational configuration of peripheral units can be established efficiently. The modules indicated to be faulty in the system map need not be (powered on and) accessed at all. On the other hand, each module indicated to be operational in the system map is powered on and tested by a strong test procedure if the module is implicated by the HSWs, and by a weak test procedure, otherwise. A peripheral module is implicated when the present execution of the recovery program was caused by one of the following events: (1) the module had reported the detection of an invalid code on the A-/D- bus without being corroborated by other units, i.e., a Cat IV fault had occurred, (2) the address that the module had loaded on the bus for reading from or writing into a memory location was an invalid code, i.e., Cat IIIa fault had occurred (but since then the A-bus has been validated), and (3) the data that the module had loaded on the D-bus for writing into a memory location was an invalid code.

The last unit to be recovered is the application memory. Recovery of the application memory consists of two steps: one to identify all the operational memory modules and the other to recover the application information (i.e., the content of the application memory that had existed prior to the fault that caused the entry of the recovery program). Similar to the establishment of an operational configuration of peripheral units, only the (application memory) modules indicated to be operational in the system map are accessed. Again a module is tested by a strong test procedure if it is implicated by the HSWs, and by a weak test procedure, otherwise. A memory module is implicated when all of the following three conditions are met: (1) the HSWs indicate the occurrence of a Cat IIId fault (i.e., detection of an invalid code on the D-bus by more than one unit) as the cause of the present execution of the recovery program; (2) the HSWs indicate that a memory read operation was in execution when the fault occurred and the address used corresponds to the module; and (3) the D-bus has been validated. If the CPU was in control of both the A-bus and the D-bus when the fault occurred, then the address used is kept in a field of a hardware status word register. On the other hand, if a peripheral module was in control of the buses, then the address used is kept in a field of a status register of the module. Recall that at the beginning of the recovery of the system memory peripheral units are powered off and the contents of their status registers are lost. Therefore, before taking power off a peripheral module that keeps the most recently used address in its status register, the recovery program must copy the content of the status register into a register within the CPU. The procedure of recovering the application information (after identifying all the operational memory modules) is essentially that described in [D1,L1].

Once all the units up to the application memory have been recovered, an acceptance test designed to check the acceptability of the recovery program execution (i.e., to check if the system has been properly recovered) is executed. This is motivated by the following considerations. First, a certain unit that has already been recovered may become faulty during recovery of other units and remain undetected until the acceptance test is executed. This is a rather minor motivation because (1) the CPU and the buses are monitored at high frequency and thus only the peripheral units and the MMU can have non- negligible latent faults, and (2) even if there is a faulty unit on resumption of normal processing, the unit can be detected later and recovered by execution of the recovery program.

A more potentially serious factor that justifies the use of the acceptance test is the possibility of incorrect recovery of a certain faulty unit by an (imperfect) recovery procedure. If the unit is detected later during normal processing, the recovery program may again incorrectly recover the unit, thus leading to a disastrous loop. This rather pessimistic but pragmatic attitude is based on recognizing the difficulty of complete understanding (or perfect analysis) of the behavior of a faulty machine. It is realistic to assume that, given any single recovery procedure, situations could arise where the procedure is not effective. It is thus a pratical necessity to make provisions in the recovery program for attempting to circumvent the situation after a recovery procedure has failed (i.e., for tolerating or covering up some local design faults/inadequacies during execution). The aforementioned loop can be broken only if a different recovery procedure is tried. The acceptance test is obviously a means of recognizing the need for execution of an alternate recovery procedure.

It is desirable that alternate recovery procedures be logically simpler and also take more global (or drastic) recovery actions than the primary recovery procedure. An alternate that we adopted, takes the following actions: (1) it recovers the CPU and the buses by use of strong tests only, (2) it uses (only) the cold restart procedure to recover the system memory, and (3) it recovers the peripheral units and the application memory again by use of strong tests only. This alternate is substantially simpler in logic than the primary although it may require a larger execution time.

The acceptance test plays an important role in increasing the robustness of the total recovery program. Two important requirements of an effective acceptance test are the simplicity and the logical independence from the primary or alternate recovery procedures. That is, the logic of an acceptance test must be simple so that the probability of having an incorrectly designed acceptance test may be minimized. In addition, it must be independent of the logic of the primary or alternate recovery procedures so that the probability of having similar errors in both the recovery procedures and the acceptance test may also be minimized. The acceptance test that we adopted, takes advantage of the following property: if the new system configuration is operational, then computation of a function that involves the use of any parts of the configuration in an arbitrary sequence must produce a correct result. It also seemed useful to take advantage of the following property: the difference between the old system configuration (i.e., the configuration prior to the execution of a recovery procedure) and the new system configuration (i.e., the configuration after the execution) must not be contradictory to the HSWs (specifically the information on the most recently reported fault). For instance, if the difference is only in the SIU configuration, then the acceptance test can check if the HSWs have implicated the (previous) SIU configuration or any unit that can be affected by a malfunctioning SIU. However, this approach was not adopted because of the difficulty of the complexity of the logic and the difficulty in saving information on an old system configuration.

If an incorrectly recovered unit is detected before the initiation of the acceptance test and followed by reinvocation of the recovery program, then a disastrous looping can still occur (because an alternate recovery program will not be executed). Since the CPU and the buses are automatically reconfigured on detection of their faults, an infinite loop due to incorrect recovery of one or both of the two units is unlikely to occur (provided that the probability of the recovery program reconfiguring an operational unit is negligible). Therefore, the problem is reduced to maximizing the probability of detecting a loop due to incorrect recovery of the system memory, peripheral units, and/or application memory. Once the system memory is recovered, recovery indicators or histories of multiple consecutive executions of the recovery program can then be recorded as long as there is storage space within the system memory. Thus if the recovery program is reentered later but before resumption of normal processing, then the recovery program can learn by examining the indicators that the present execution is not the first execution since the interruption of most recent normal processing. (In a sense, flag 1 serves the function of a recovery indicator to a limited extent.) After some iterations the recovery program can revert to an alternate recovery procedure.

7. <u>Specification of a recovery program in PASCAL</u>

The procedure discussed in the preceding section was specified in PASCAL. Appendix B contains a complete listing of the PASCAL specification. The specification is in sufficient detail so that a final machine program can be obtained in a straightforward manner from it. Some low-level procedures were specified in plain words (enclosed by '<' and '>') because their PASCAL specification was not expected to be more readable while their functions are simple.

There are two types of variables declared in the PASCAL specification. One set of variables called system variables represent storage components in the FTSC, i.e., registers and memory words. The other set of variables are program variables that exist only during execution of a recovery program. Each module is abstracted into the set of registers and functional capabilities (including error detection) that the module possesses. The abstraction of each type of module is treated as a data type in the specification. Due to the restriction in PASCAL, descriptions of functional capabilities of each module were introduced as comments.

The procedures in the PASCAL specification are organized as follows. The main procedure carries out the sequence of functional unit recoveries as discussed in the preceding section. The main procedure calls other procedures which are divided into two classes; the procedures of testing modules form one class while the remaining procedures form the other class. Comments were inserted in various places in order to make the entire program self-explanatory. The objective of each procedure/function is explained either immediately after the procedure/function statement or where it is called (i.e., inside the body of a calling procedure).

8. Assembly language coding of the PASCAL specification

Assembly language coding of the PASCAL specification can be done in a straightforward manner. For example, let us consider the following statement extracted from the beginning portion of the main procedure.

```
for i:=5 to 8 do
   if hsw1.fault_cat[i]=1 then begin pflag6; pflag7 end;
```

This statement is designed to test the CPU's capability of reading HSW1. This can be coded as follows.

```
         LDR,2   =4         (set the iteration counter)
         LDR,1   X'F800'    (load reg1 with HSW1)
         LRS,1   =-3        (shift left by 3 positions)
[396]:   LRS,1   =-1
         JPZ,1   400        (skip if the left-most bit is 0)
         STZ     X'F80E'    (program flag 6)
         STZ     X'F807'    (program flag 7)
[400]:   JDN,2   396        (repeat if 4 iterations are not done)
```

The structure of the PASCAL specification should be preserved in assembly language coding to reduce the cost of debugging and maintenance.

The recovery block in the PASCAL specification does not require saving any register or memory word at the beginning or in the middle of execution of the recovery program. It was designed in that form to preserve the simplicity of the overall recovery program. Thus coding the recovery block in an assembly language should again be straightforward.

9. Summary

Design of an effective recovery program for a fault-tolerant computer such as the FTSC represents a special type of challenge in software engineering. The difficulty in this design is related to the unusual input to this program, i.e., a faulty machine whose behavior is difficult to analyze. In view of the vital role that the recovery program plays in fault-tolerant computing, it must be designed with simple (i.e., easy to understand) and systematic procedures of recovering the system. The rules of fixed-order unit recovery and conservative recovery discussed in section 4 meet these criteria. Recognizing the difficulty of complete understanding of the behavior of a faulty machine, it seems also realistic to provide an alternate recovery procedure that can be used when the primary is not effective, and the acceptance test that can judge the effectiveness of a recovery procedure at run-time. Measures of success in this direction largely depends upon the logical simplicity of the acceptance test and the logical independence among two recovery procedures and the acceptance test that are used.

In order to obtain a well-structured recovery program and also take advantage of available tools (applicable to high level language programs), the approach of first obtaining the high level language specification of a recovery program instead of directly constructing a machine/assembly language recovery program was adopted. PASCAL was chosen as the specification language. The PASCAL specification is almost as precise as the machine program and yet almost as readable as a natural language specification. It can also be checked out to a certain extent by using a PASCAL compiler, provided that various actions of the FTSC are simulated by (PASCAL) procedures. Such a checkout is thus limited by the simulation cost.

Although much care has been taken in this development to obtain an easily understandable recovery program and the checkout of the program with a PASCAL simulator-compiler might be of help in establishing confidence in the correctness of the PASCAL specification, it seems worthwhile and desirable to further enhance the confidence through supplementary means. The application of a program verification approach to this recovery program seems well justified, considering the relatively small program size and yet the important role of the recovery program. It does not appear that existing verification approaches can be directly applied to the recovery program, though. It seems that the existing approaches need to be modified or extended, partly due to the fact that the faults (which are inputs to the recovery program) are non-deterministic events. This remains as a future research subject.

References

[B1] Burchby, D., Kerr, L.W. and Sturm, W.A., "Specification of the Fault-Tolerant Spaceborne Computer (FTSC)", Proc. 1976 Int'l Symp. on Fault-Tolerant Computing, pp.129-133.

[D1] DeAngelis, D. and Lauro, J.A., "Software recovery in the Fault-Tolerant Spaceborne Computer", Proc. 1976 Int'l Symp. on Fault-Tolerant Computing, pp.143-147.

[F1] Fanelli, E.V. and Hecht, H., "The Fault-Tolerant Spaceborne Computer", Proc. 2nd AIAA Digital Avionics Systems Conf., 1977, pp.73-80.

[H1] Hecht, H., "Fault-tolerant software for real-time applications", Computing Surveys, Dec. 1976, pp.391-407.

[H2] Hecht, H. and Stiffler, J.J., "Redundancy allocation in the Fault-Tolerant Spaceborne Computer", Proc. 1978 Int'l Symp. on Fault-Tolerant Computing, paper 6a.4.

[H3] Horning, J.J. et al, "A program structure for error detection and recovery", Lecture Notes in Comp. Sci., vol. 16, Springer-Verlag, 1974, pp.171-187.

[K1] Kim, K.H. and Ramamoorthy, C.V., "Recent developments in software fault tolerance through program redundancy", Proc. 10th Hawaii Int'l Conf. on System Sciences, Jan. 1977, pp.234-238.

[K2] Kim, K.H. and Arshi, T., "A translator of PASCAL augmented with recovery block", Part II of this final report (U.S. Air Force - SAMSO contract F04701-77-C-0120).

[L1] Logicon, Inc. and Raytheon Co., 'Brassboard Fault Tolerant Spaceborne Computer (BFTSC): operating system maintenance manual' CDRL-A017, Nov. 1976.

[O1] O'Brien, F.J., "Rollback Point Insertion Strategies", Proc. 1976 Int'l Symp. on Fault-Tolerant Computing, pp.138-142.

[R1] Randell, B., "System structure for software fault tolerance", IEEE Trans. on Software Engineering, June 1975, pp.220-232.

[S1] SAMSO YAD, 'Development specification for the Fault Tolerant Spaceborne Computer (FTSC)' preliminary version, 1977.

[S2] Stiffler, J.J., "Architectural design for near-100 % fault coverage", Proc. 1976 Int'l Symp. on Fault-Tolerant Computing, pp.134-137.

Appendix A: Syntax and semantics of a recovery block

A recovery block (RB) has the following syntactic structure:

```
ensure T
     by O1
     else-by O2

     ----------

     else-by On
else-error
```

where T denotes the acceptance test, O1 the primary object block, and Ok (2_<k_<n) the alternate object blocks.

All the object blocks in an RB specify computations aimed at producing the same or approximately the same result. A process executes the acceptance test T on exit from an object block to confirm that the result of the object block execution is acceptable. If it is acceptable, the process exits from the RB. If it is not, the process enters the next alternate object block. Also, the process enters the next alternate object block if the underlying processor system detects an error (e.g., divide-by-zero) while the process is inside an object block.

Before an alternate object block is entered, the process state is restored to the state that existed just before entry to the primary object block. That is, the process rolls back to the recovery point (RP) established on entry to the RB. Each variable that was assigned a new value by the rejected execution is restored to its original value. The underlying processor system automatically performs this "assignment reversal". To enable this, the first assignment to a non- local variable v during execution of an object block is preceded by the recording of the original value of v, denoted by PRIOR(v), in the recovery cache. Actually PRIOR(v) may also be used within the acceptance test of the RB. These (first) assignment records need to be kept until the RB is successfully exited. When an RB is exited, the RP established on entry to the RB may be discarded.

Appendix B: PASCAL specification of an FTSC recovery program

```
const faulty    =0;
      working   =1;
      yes       =1;
      no        =0;
      lastmodid =59;          (*no of modules*)
      dummy     =0;
      cat1      =15;
      cat2      =12;
      cat3a     =8;
      cat3b     =9;
      cat4a     =3;
      cat4b     =1;
      cat4c     =2;
      cat4d     =6;
      cat4e     =7;
      mmuid     =1;
      siuid     =2;
      dma1id    =3;
      dma2id    =4;
      cpuid     =5;
      abusid    =6;
      dbusid    =7;
      ccuid     =8;
      htid      =9;
      cuid      =10;
      tuid      =11;
      puid      =12;
      mmusize   =24;
      bustestreptition = 8;

type  bit      =(0,1);
      register=array[0..31] of bit;
      word     =array[0..40] of bit;
      reg2     =array[0..1]  of bit;
      reg3     =array[0..2]  of bit;
      reg4     =array[0..3]  of bit;
      (*monitor=procedure*)

(*------------*)
(*module types*)
(*------------*)

      cpumodule=(*module*)

            record
               hsw1 : record
                         flag:array[1..3] of bit;
                                 (*actual bit position: 0..2*)
                         alexicindicator:bit;
                         fault_cat:array[5..8] of bit;
                         statechange:bit;
                         cputest:bit;
```

```
                    simplexmode:bit;
                    busarbiter:array[12..13] of bit;
                    readstate:bit;
                    softaddr:bit;
                    mrar:array[16..31] of bit
                 end
          hsw2 : record
                    dbusspare:array[0..2] of bit;
                    abusspare:array[3..4] of bit;
                    acpuid:array[5..6] of bit;
                    mcpuid:array[7..8] of bit;
                    pustate:array[9..12] of bit;
                    tustate:array[13..16] of bit;
                    rti:array[17..19] of bit;
                 end
         mcpumask: register;
         intrmask: register;
         gpreg: array[0..7] of register;
       end(*-record*);

       (*operations
         pflag0,pflag1,pflag2,pflag3,
         pflag4,pflag5,pflag6,pflag7:regular ;
         buscodmon: monitor;(*bus code monitor*)
         wdt      : monitor;(*watchdog timer*)
         ilopdet  : monitor;(*illegal opcode detector*)
         ctrlcomp : monitor;(*control comparator*)
         adcomp   : monitor;(*address and data comparator*)
       end-operations*)

     (*end-module*)

mmumodule=(*module*)

       record
         status     : register;
         oldriphigh: register;
         oldriplow : register;
         newriphigh: register;
         newriplow : register;
         soft1      : reg4     ;
         soft2      : reg4     ;
         syndrom    : reg6     ;
         wrprotreg : reg4     ;
         content    : array[0..4093] of word;
         nexttolastword :word;
         lastword  : word     ;
         (*contents of the memory words*)
         modtype    :integer  ;(*11-application memory module
                                 00-system memory module 0
                                 01-system memory module 1*)
         sysmodcontent=
            record
               hardname:integer;
               softtable:array[0..14] of
```

```
                                      record
                                        hardname:array[27..31] of bit;
                                        lost:bit;
                                      end
                        ppmask= record
                                     dma1mask:array[26..27] of bit;
                                     dma2mask:array[28..29] of bit;
                                     siumask :array[30..31] of bit;
                                end
                        hardtable:array[1..24] of
                                      record
                                        activespare:  bit;
                                        dupinfstart: array[0..14] of
                                                     integer;
                                        dupinfend   : array[0..14] of
                                                     integer;
                                        softname:array[28..31] of bit;
                                        ripplerconfiguration:
                                             array[0..40] of bit;
                                        sparebit    : boolean
                                      end
                        applicationmmu: integer;
                        transcount:array[0..lastmodid] of integer;
                        constlost :array[0..14] of bit;
                        lostunit  :set of integer;
                     end
                end(*-record*);

                (*operations
                  datacodmon: monitor; (*data code monitor*)
                  buscodmon : monitor; (*bus code monitor*)
                  analogmon : monitor; (*analog monitor*)
                  syndmon   : monitor; (*syndrom monitor*)
                  softmon   : monitor; (*softname monitor*)
                  wrprotmon : monitor; (*write protect monitor*)
                  addrslmon : monitor; (*address select monitor*)
                end-operations*)

              (*end-module*)

    ccumodule=(*module*)

                record
                  faultreg:register;
                  abusspid: reg2;
                  dbusspid: reg3;
                  acpuid: reg2;
                  mcpuid: reg2;
                  flag1,flag2,flag3:bit ;
                end(*-record*);
```

```
          (*operations
            pflagcomp : monitor;  (*program flag comparator*)
            wdt       : monitor;
            faulthand : regular;  (*fault handler*)
          end-operations*)

        (*end-module*)

dmamodule=(*module*)

          record
            cost0     = record    (*control/status register 0*)
                           mrar:array[16..31] of bit;
                        end
            cost1     : register; (*control/status register 1*)
          end(*-record*);

          (*operations
            buscodmon : monitor; (*bus code monitor*)
            eobdet    : monitor; (*end of block detector*)
            wdt       : monitor; (*watchdog timer*)
            ifm       : monitor; (*internal fault monitor*)
          end-operations*)

        (*end-module*)

siumodule=(*module*)

          record
            cost0     = record
                           mrar:array[16..31] of bit;
                        end
            cost1      :register;
          end(*-record*);

          (*operations
            buscodmon : monitor;
            eobdet    : monitor;
            wdt       : monitor;
            seqdbusmon: monitor;(*sequential data bus monitor*)
            synchdet  : regular;
          end-operations*)

        (*end-module*)

pumodule =(*module*)

          record
            status1   : bit;
            status2   : bit;
          end(*-record*);
```

```
                (*operations
                  involmon : monitor; (*input voltage monitor*)
                  outvolmon: monitor; (*output voltage monitor*)
                end-operations*)

              (*end-module*)

cumodule =(*module*)

                record
                  ccuclamp : bit;
                  mmuclamp : bit;
                  iohtclamp: bit;
                end(*-record*);

                (*operations
                  raddet    : monitor;
                end-operation*)

              (*end-module*)

tumodule =(*module*)

                record
                  status1 : bit;
                  status2 : bit;
                end(*-record*);

                (*operations
                  voltmon : monitor;
                  clkmon1 : monitor;
                  clkmon2 : monitor;
                  rtimon1 : monitor;
                  rtimon2 : monitor;
                end-operations*)

              (*end-module*)

htmodule =(*module*)

                record
                  htstatus : bit;
                end(*-record*);

                (*operations
                  comparator:register;
                end-operations*)

              (*end-module*)
```

```
var

(*system variables*)
        cpu    : array[0..3]  of cpumodule;
        mmu    : array[0..mmusize] of mmumodule;
        dma1   : array[0..1]  of dmamodule;
        dma2   : array[0..1]  of dmamodule;
        siu    : array[0..1]  of siumodule;
        pu     : array[0..1]  of pumodule;
        tu     : array[0..1]  of tumodule;
        ccu    :                 ccumodule;
        cu     :                 cumodule;
        ht     :                 htmodule;


(*program variables*)
        status      : array[0..lastmodid] of bit;
        indictedset : set of integer; (*softname*)
        reconfcount : integer;
        sysmodid: array[0..1] of integer;
        temptrans: array[0..lastmodid] of bit; (*register*)
        numfound: integer;
        sparemmm: integer;
        lastusedmmm: integer;
        i: integer;
        j: integer;
        k: integer;

(*----------------------------*)
(*utility procedures/functions*)
(*----------------------------*)

function faultcat:integer;
(*this translates a 4-bit vector cpu[ccu.acpuid].hsw1.fault_cat
    into a decimal faultcat number*)
begin
   with cpu[ccu.acpuid].hsw1 do
   faultcat:=bintodeci(0,0,0,0,fault_cat[5],fault_cat[6],fault_cat[7],
            fault_cat[8])
end

function id(var unitid,moduleno:integer):integer;
begin
   case unitid of
      1:(*mmu*)    id:= 1+moduleno;
      2:(*siu*)    id:=25+moduleno;
      3:(*dma1*)   id:=27+moduleno;
      4:(*dma2*)   id:=29+moduleno;
      5:(*cpu*)    id:=32+moduleno;
      6:(*abus*)   id:=36+moduleno;
      7:(*dbus*)   id:=40+moduleno;
      8:(*ccu*)    id:=46+moduleno;
      9:(*ht*)     id:=49+moduleno;
     10:(*cu*)     id:=52+moduleno;
     11:(*tu*)     id:=55+moduleno;
```

```
      12:(*pu*)      id:=57+moduleno;
   end (*case*)
end

procedure settemptrans;
(*marks the occurrence of a transient fault in a temporary counter
   register*)
begin
   case faultcat of
      12  :  begin  (*cat2*)
                 temptrans[id(cpuid,acpuid)]:=1;
                 temptrans[id(cpuid,mcpuid)]:=1;
                 temptrans[id(abusid,abusspid)]:=1;
                 temptrans[id(dbusid,dbusspid)]:=1;
             end
      8   : begin  (*cat3a*)
                 temptrans[id(cpuid,acpuid)]:=1;
                 temptrans[id(cpuid,mcpuid)]:=1;
                 temptrans[id(abusid,abusspid)]:=1;
             end
      9   : begin  (*cat3b*)
                 temptrans[id(cpuid,acpuid)]:=1;
                 temptrans[id(cpuid,mcpuid)]:=1;
                 temptrans[id(dbusid,dbusspid)]:=1;
             end
   end (*case*)
end


function lastbususer:integer;
(*this converts cpu[ccu.acpuid].hsw1.busarbiter into a unitid*)
begin
   with cpu[ccu.acpuid] do
   lastbususer:=bintodeci(0,0,0,0,0,busarbiter[12],busarbiter[13])+2
end


function ppusoftname:integer;
(*identify the lastbususer*)
begin
   case lastbususer of
      0:ppusoftname:=19 (*cpu*);
      1:ppusoftname:=18 (*siu*);
      2:ppusoftname:=16 (*dma1*);
      3:ppusoftname:=17 (*dma2*);
   end (*case*)
end


function  bintodeci(var bit0,bit1,bit2,bit3,bit4,bit5,bit6,bit7:bit):
                                                              integer;
(*this translates an 8-bit vector into an equivalent moduleno*)
begin
   bintodeci:=bit7+bit6*2+bit5*2'2+bit4*2'3+bit3*2'4+bit2*2'5+bit1*2'6
                +bit0*2'7
```

```
end


function  suspected:integer;
(*converts the right most 16-bit of hsw1 into a decimal integer*)
begin
   with cpu[ccu.acpuid].hsw1 do
   suspected:=bintodeci(0,0,mrar[16],mrar[17],mrar[18],mrar[19]);
   if suspected=15 then
   case bintodeci(0,0,0,0,mrar[28],mrar[29],mrar[30],mrar[31]) of
      0:suspected:=16 (*dma1*);
      1:suspected:=16 (*dma1*);
      2:suspected:=17 (*dma2*);
      3:suspected:=17 (*dma2*);
      4:suspected:=18 (*siu*);
      5:suspected:=18 (*siu*);
   end (*case*)
end


function activeppmid(var unitid:integer):integer;
(*returns the id of the active module of a given PPU*)
begin
   with mmu[sysmodid[0]].sysmodcontent.ppmask do
   case unitid of
      3:begin
           if bintodeci(0,0,0,0,0,0,dma1mask[26],dma1mask[27])=1
              then activeppmid(dma1id):=0
           else activeppmid(dma1id):=1
        end
      4:begin
           if bintodeci(0,0,0,0,0,0,dma2mask[28],dma2mask[29])=1
              then activeppmid(dma2id):=0
           else activeppmid(dma2id):=1
        end
      2:begin
           if bintodeci(0,0,0,0,0,0,siumask[30],siumask[31])=1
              then activeppmid(siuid):=0
           else activeppmid(siuid):=1
        end
   end (*case*)
end


function mrusedmmm :integer;
begin
   case lastbususer of
      dma1id:begin
                with dma1[activeppmid(dmaid)].cost0  do
                mrusedmmm:=bintodeci(0,0,0,0,mrar[16],mrar[17],
                                                mrar[18],mrar[19])
             end
      dma2id:begin
                with dma2[activeppmid(dma2id)].cost0  do
                mrusedmmm:=bintodeci(0,0,0,0,mrar[16],mrar[17],
                                                  mrar[18],mrar[19])
              end
       siuid: begin
                 with siu [activeppmid(siuid) ].cost0  do
                 mrusedmmm:=bintodeci(0,0,0,0,mrar[16],mrar[17],
                                                  mrar[18],mrar[19])
              end
    end (*case*)
end


function ismapflag(var hardmodname,numfound:integer):boolean;
var onecounter:integer;
begin
   with mmu[hardmodname] do
   begin
      onecounter:=0;
      soft1:=numfound;
      soft2:=numfound;
      for i:=0 to 39 do
      if lastword[i]:= 1 then onecounter:=onecounter+1
   end
   if onecounter> 24 or onecounter=24 then
      ismapflag:=yes;
   else ismapflag:=no
end


procedure findsysmodid(var numfound:integer;sysmod[0..1]:array
                              of integer)
(* This procedure finds the mmu modules containing  the  system  map.
"numfound"  is  an  integer  which  represents  the  number of system
modules found.  System[i] represents the  hard  name  of  the  system
module found, where i is either 0 or 1 corresponding to the first
or the second system memory module found*)
(* instructions "power on", "power off","ignore read", and
"ignore write" are assumed to be working correctly *)
var hardmodname:integer;
begin
   numfound:=0;
   hardmodname:=0;
   sysmod[0]:=0;
   sysmod[1]:=0;
   repeat
      hardmodname:=hardmodname+1;
      poweron(hardmodname);   (* turn on the power of the mmu
                              module with hard name "hardmodname"*)
      if ismapflag(mmu[hardmodname],numfound) then
         (* this will determine if the mmu module with hard name
         = "hardmodname", containes a system map.  numfound is a
         dummy soft name that is temporarily stored into the soft
         name registers of the corresponding memory module*)
      begin
          sysmod[numfound]:=hardmodname;
          numfound:=numfound+1
      end(*then*)
      else poweroff(mmuid,hardmodname);
   until hardmodname=24 or numfound=2;
end

procedure coldrestart;
var i:integer;
begin
   (*find two good mmu modules*)
   i:=-1; j:=0;
   repeat begin
     i:=i+1;
     poweron(mmuid,i);
     soft1[i]:=j; soft2[2]:=j;
     if mmu_hardval(j) then begin sysmod[j]:=i; j:=j+1 end
   until j=2 or i=mmusize;
   finddma; (*find one good dma module*)
   loadsystmem; (*initialize the system memory using the information
                  in the backup memory*)
   findppu; (*find one operational module of each ppu*)
   recordppu; (*record the mod ids of operational ppu modules into
                the system map*)
   findmmu; (*find all the operational mmu modules*)
   recordmmu; (*record the mod ids of operational mmu modules into
                the system map*)
   loadapplmem; (*initialize the application memory using the
                  information in the backup memory*)
end

procedure makecopy(var softnam1,hardnam1,softnam2,hardnam2:integer);
begin
   hardnam2:=findspare;
   if hardnam2=0  then hardnam2:=findapplmemmod (*find an application
          memory module to be converted into a system memory module*)
   with mmu[hardnam2] do
   begin
      soft1:=softnam2; soft2:=softnam2
   end
   copydupinfo((*from*) softnam1,hardnam1, (*to*) softnam2,hardnam2);
end

procedure updatesysmap(var unitid,activeppmid(unitid));
begin
   for i:=0 to 1 do
      with mmu[sysmodid[i]].sysmodcontent do
      begin
         case unitid of
         dma1id: j:=26;
         dma2id: j:=28;
         siuid : j:=30
         end(*case*);
         if status[unitid,(activeppmid(unitid)+1) mod 2]
            = working then
         begin
            if (activeppmid(unitid)+1) mod 2 = 0 then
            begin
               ppmask(j+1):=1;
               ppmask(j):=0
            end(*-then*)
            else begin
               ppmask(j+1):=0;
               ppmask(j):=1
            end(*-else*)
         end(*-then*)
         else begin
               ppmask(j+1):=1
               ppmask(j):=1;
         end(*-else*)
      end
end(*-procedure*)

procedure copydupinfo(var softnam1,hardnam1,softnam2,hardnam2:integer);
(*copies duplicate information from one module to another*)
begin
   with mmu[sysmodid[0]].sysmodcontent do
   for i:=dupinfstart to dupinfend do
   begin
      read(mmu[hardnam1].content[i],cpu[ccu.acpuid].gpreg[1]);
      write(cpu[ccu.acpuid].gpreg[1],mmu[hardnam2].content[i])
   end
end

function mmmsoftname(var hardname:integer):integer;
(*find the main memory module softname with the given hardname*)
begin
   with mmu[sysmodid[0]].hardtable[hardname] do
   mmmsoftname:=bintodeci(0,0,0,0,softname[28],softname[29],
                      softname[30],softname[31])
end

function mmmhardname(var softname:integer):integer;
(*find the main memory module hardname with the given softname*)
begin
   with mmu[sysmodid[0]].sysmodcontent.softtable[softname] do
   mmmhardname:=bintodeci(0,0,0,hardname[27],hardname[28],
                       hardname[29],hardname[30],hardname[31])
end

procedure setcat4trans(var unitid,moduleno:integer);
begin
   case faultcat of
      cat3a: begin
                temptrans[id(abusid,abusspid)]:=1;
                temptrans[unitid]:=1
              end
      cat3b: begin
                temptrans[id(dbusid,dbusspid)]:=1;
                temptrans[unitid]:=1
             end
      cat4:  temptrans[unitid]:=1
   end(*-case*)
end(*-procedure*)

function arbitlocation:integer;
<returns a random integer within the interval 0 to 4095>
end

procedure pp_test(unitid:integer,hardval:bit):boolean;
begin
   poweron(unitid,activeppmid(unitid));
   case hardval of
   1: if pp_hardval(unitid) then
      begin
         setcat4trans(unitid);
         status[id(unitid,activeppmid(unitid))]:=working
      end
   0: if pp_weakval(unitid) then
      status[id(unitid,activeppmid(unitid))]:=working
   end(*case*)
   if status[id(unitid,activeppmid(unitid))]=faulty then
   begin
      poweroff(unitid,activeppmid(unitid));
      poweron(unitid,(activeppmid(unitid)+1) mod 2);
      if pp_hardval(unitid) then
      status[id(unitid,(activeppmid(unitid)+1) mod 2]:=working
      else begin
         poweroff(unitid,(activeppmid(unitid)+1) mod 2);
         with mmu[sysmodid[0]].sysmodcontent do
         lostunit:=lostunit+[unitid]
      end(*else*);
      updatesysmap(unitid);
      updateppmask(unitid)
   end
end

function findspare:integer;
begin
   with mmu[sysmodid[0]].sysmodcontent do
   begin
      i:=1;
      findspare:=0;
      repeat
         i=i+1;
         if hardtable[i].activespare=1 (*spare*) then
         begin
            poweron(mmuid,i);
            soft1[i]:=15; soft2[i]:=15;
            if mmu_hardval(15) then
            begin
               findspare:=i;
               hardtable[i].activespare:=0 (*active*)
            end
            else poweroff(mmuid,i)
         end
      until i>23 or findspare<>0
   end
end

procedure updatetranscount(iounitid,iomodid);
(*this procedure increments the transient faults as follows.*)
begin
   for j:=1 to lastmodid do begin
      for i:=0 to 1 do
         with mmu[sysmodid[i]].sysmodcontent do
         transcount[i]:=transcount[i]+temptrans[j];
      if mmu[sysmodid[0]].sysmodcontent.transcount[i]>limittranscount
         then switch_module (*transcount[i] is reset, a spare module
                                is turned on and then tested*)
   end
end

procedure reconfsysmod(softnam1,softnam2:integer);
begin
   poweron(mmuid,sysmodid[softnam2]);
   if refreshable(softnam2,sysmodid[softnam2]) then
   begin
      refresh(softnam2,sysmodid[softnam2]);
      status[id(mmuid,sysmodid[softnam2])]:=working;
      copydupinfo(softnam1,sysmodid[softnam1],softnam2,
                sysmodid[softnam2])
   end(*then*)
   else begin
      makecopy(softnam1,sysmodid[softnam1],softnam2,sysmodid[softnam2]);
      status[id(mmuid,sysmodid[softnam2])]:=working;
      constlost[softnam2]:=yes
   end(*else*);
   record(softnam2,sysmodid[softnam2])
end

procedure refreconf(softnam1,softnam2:integer);
begin
   refresh(softnam1,sysmodid[softnam1]);
   status[id(mmuid,sysmodid[softnam1])]:=working;
   poweron(mmuid,sysmod[softnam2]);
   if refreshable(softnam2,sysmodid[softnam2]) then
   begin
      refresh(softnam2,sysmodid[softnam2]);
      status[id(mmuid,sysmodid[softnam2])]:=working
   end(*then*)
   else begin
      makecopy(softnam1,sysmodid[softnam1],softnam2,sysmodid
      constlost[softnam2]:=yes;
      status[id(mmuid,sysmodid[softnam2])]:=working
   end(*else*);
   record(0,sysmod[0]);
   record(1,sysmod[1])
end
```

```
function acceptable:boolean;
begin
   <prepare an arbitrary nuamber x in gpreg- (general purpose register)
    1 and write x into an arbitrary location y of a randomly chosen
    module>;
   <load x in y into gpreg-2>;
   <square gpreg-2 and store the result into an arbitrary location
    p of a randomly chosen module q>;
   <load the content of p into gpreg-3>;
   <if gpreg-1 * gpreg-2 <> gpreg-3 then acceptable := yes
    else acceptable := no>;
end

(*----------------------------*)
(*   module test procedures   *)
(*----------------------------*)

(* 1. cpu test *)

function  cpu_hardval:boolean;
   begin
      (*part common to both acpu & mcpu*)
      gpreg[1]:=(gpreg[1]+gpreg[2])*gpreg[3] - (gpreg[1]*gpreg[3]
                             +gpreg[2]*gpreg[3]);
      if gprerg[1]=0 then begin pflag6; pflag7 end;
      (*mcpu comparator test*)
      for i=24 to 31 do
      begin
         with cpu[ccu.acpu] do
         set(mcpumask[i]) (set bit position i of mcpu mask register*)
      end;
      <bus code monitor test>
   end

function cpu_weakval:boolean;
begin
   if gpreg[1]+gpreg[2]>0 then begin pflag6; pflag7 end;
   with cpu[ccu.acpu] do set(mcpumask[i])
end

(* 2. bus test *)

function bus_val:boolean; (* use the following three test patterns
                    to put one's and zero's on each wire of bus.
                    monitors in cpu will detect the bus fault.*)
begin
   with ccu do
   begin
      pflag6;
      for i:=1 to bustestreptition do
      begin
         storehard('0'(*module's hardname to be stored to*),
                '0000'(*abus*),'00000000'(*dbus*));
         if flag2=0 then
         begin bus_val:=no; go to 100 end
```

```
            storehard('0','FFAB','FFABFFFF');
            if flag2=0 then
            begin bus_val:=no; go to 100 end
            storehard('0','F9F4','F9F4FFFF');
            if flag2=0 then
            begin bus_val:=no; go to 100 end
         end;
         bus_val:=yes
      end
      100:
   end (*procedure*)

   (* 3. memory module test *)

   function mmu_weakval(var softname,hardname:integer);
   var location:integer;
   begin
      with mmu[hardname] do
      with cpu[ccu.acpuid] do
      begin
         storerippler(softname,hardname);
         location:=arbitlocation; (*random location between 0 and 4095*)
         read(content[location],gpreg[1]);
         write(gpreg[1],content[location]);
         read(content[location],gpreg[2]);
         if gpreg[1]=gpreg[2] then
            readwrite_val:=yes
         else readwrite_val:=no
      end
   end

   function mmu_hardval(var softname,hardname:integer):boolean;
   begin
      storerippler(softname,hardname);
      if (readwrite_val(hardname) and
         datacodmon_val(hardname) and
         addrcodmon_val(hardname) and
         buscodmon_val(hardname) and
         syndmon_val(hardname) and
         softnamemon_val(hardname) and
         writeprotmon_val(hardname) and
         analogmon_val(hardname) and
         addrselmon_val(hardname)) then
         sysmod_val:=yes
      else sysmod_val:=no
   end

   (* 4. peripheral test *)

   function pp_weakval(var ppid:integer):boolean;
   begin
      (*send a dummy command through  peripheral  to  the  corresponding
      control word.  *)
      case ppid of
         2:(*siu *) store('F845'(*control word 1 in siu*),'command');
      3:(*dma1*) store('F841','command');
      4:(*dma2*) store('F843','command');
   end;
   if endofblock(ppid) then
      pp_weakval:=yes
   else pp_weakval:=no
end

function pp_hardval(var ppid:integer):boolean;
         function commonpart_val(ppid:integer);
         begin
            if pp_weakval(ppid) and
               wdt(ppid) and
               ifm_val(ppid) (*internal fault monitor*) and
               buscodmon_val(ppid) then
               commonpart_val:=yes
            else commonpart_val:=no
         end
begin
   case ppid of
      2:if commonpart_val and seqbuscodmon_val then
           pp_hardval:=yes
        else pp_hardval:=no;
      3:pp_hardval:=commonpart_val;
      4:pp_hardval:=cpmmonpart_val
   end
end

(*-----------------------------------------------------------*)
(*functional and monitor test procedures within an mmu module*)
(*-----------------------------------------------------------*)

(*address decoder test procedure*)

function addrcodmon_val(var moduleid:integer):boolean;
begin
   pflag5; (*set flag 2*)
   ezpara;
     (*arm the enabl zero address parity hard address function which
     forces zero parity as the input address parity*)
   read(mmu[moduleid].arbtword,cpu[ccu.acpu].gpreg[-2]);
     (*after the read action, if the address decoder is working
     properly, a fault interrupt would have been generated and ccu
     flag 2 reset.  *)
   if ccu.flag2=0 then
      addrcodmon_val:=yes
   else begin
      pflag7  ;  (*reset flag 2*)
      addrcodmon_val:=no
   end;
   amreset(moduleid)
end
```

```
(*write protect monitor test procedure*)

function wrprotmon_val(var moduleid:integer):boolean;
begin
   pflag5 ;
   ldwp(moduleid,arbtword)  ;
     (*establish a write protect region associated with the location of
     arbtword, using the load write-protect hard address function*)
   write(mmu[moduleid].arbtword,cpu[ccu.acpu].gpreg[1])  ;
     (*if the region is protected, a fault interrupt would have been
     caused by the write action and ccu flag 2 reset*)
   if ccu.flag2=0 then
      wrprotmon_val:=yes
   else begin
      pflag7  ;   (*reset flag 2*)
      wrprotmon_val:=no
   end;
   amreset(moduleid)
end

(*soft name monitor validatin procedure*)

function softmon_val(var moduleid:integer)boolean;
begin
   pflag5  ;
   ldsftn(moduleid)  ;
     (*establish different soft names in the two soft name registers
     with the set soft name hard address function *)
   read(mmu[moduleid].arbtword,cpu[ccu.acpu].gpreg[0]);
     (*if the soft name monitor works properly, a fault interrupt would
     have been generated by the read action and the ccu flag2 reset*)
   if ccu.flag2=0 then
      softmon_val:=yes
   else begin
      pflag7  ;   (*reset flag 2*)
      softmon_val:=no
   end;
   amreset(moduleid)
end

(*procedure for data decoder test*)

function datacodmon_val(var moduleid:integer):integer;
begin
   pflag5  ;
   read(mmu[moduleid].arbtword,cpu[ccu.acpu].gpreg[2]);
   sbpdo(badword)  ;
     (*use the store with bad parity data instruction to put the
     improperly coded information on the bus*)
   pflag5  ;
   ebadpard(mmu[moduleid].arbtword)  ;
     (* arm the mmu enable bad patity hard address function
     to permit the information to be stored*)
   read(mmu[moduleid].arbtword,cpu[ccu.acpu].gpreg[3])  ;
     (*if data code monitor works properly, above read action
      would have caused a fault interrupt and reset ccu flag 2*)
    if ccu.flag2=0 then
       datacodmon_val:=yes
    else begin
       pflag7  ;    (*reset flag 2*)
       datacodmon_val:=no
    end;
    amreset(moduleid);
    write(mmu[moduleid].arbtword,cpu[ccu.cpu].gpreg[2])
      (*put the original word back to the same location*)
end

(*------------------------------*)
(* begining of the main program *)
(*------------------------------*)

begin
   with ccu do
   with cpu[acpuid] do
   ensure acceptable (*acceptance test*)
   by

(*--------------------------*)
(*primary recovery procedure*)
(*--------------------------*)

   begin

(*validation of cpu's capability of reading HSW1*)

      for i:=5 to 8  do
         if hsw1.fault_cat[i]=1 then begin pflag6; pflag7 end;

    (*if one of the cpu modules is faulty, then the ccu may receive
    a program flag from only one cpu module*)

(*initialization: clear status, temptrans, & indictedmoduleset*)

      for i:=0 to lastmodid do  begin
                                   status[i]:=faulty;
                                   temptrans[i]:=0
                                end
      skip3:=no;
      indictedset:=[];
      for i:=0 to 14 do
      constlost[i]:=no;

(*trust all TMR units and other automatically recovering units*)

(*CPU test*)
      if faultcat=cat2 then
      begin
         if cpu_hardval then
         begin
            status[id(cpuid,acpuid)]:=working;
            status[id(cpuid,mcpuid)]:=working;
            if hsw1.statechange = no then
            begin
               if bus_val then
               begin
                  status[id(abusid,abusspid)]:=working;
                  status[id(dbusid,dbusspid)]:=working;
                  skip3:=yes;
                  settemptrans
               end
            end
         end
      (*failure in cpu-hardval will result in a cpu reconfiguration
        and reentry of the recovery program*)
      end
      else
         if cpu_weakval then
         begin
            status[id(cpuid,acpuid)]:=working;
            status[id(cpuid,mcpuid)]:=working
         end;

(*bus test*)
      if skip3=no then
         begin
            reconfcount:=0;
            while not bus_val do
            begin
               reconfcount:=reconfcount+1;
               if reconfcount>23 then
                  cycle_cpu  (*this will cause a fault interrupt*)
               else begin
                  pflag6;
                  cycle_abus;
                  cycle_dbus;
                  pflag7
               end
            end
            status[id(abusid,abusspid)]:=working;
            if (hsw1.statechange=no and faultcat=cat3a and
                reconfcount=0) then
            begin
               if lastbususer=cpuid then settemptrans
               else indictedset:=indictedset
                                +[ppusoftname(*lastbususer*)]
            end;
            status[id(dbusid,dbusspid)]:=working;
            if (hsw1.statechange=no and faultcat=cat3b and
               reconfcount=0) then
               case hsw1.readstate of
               1(*read*):indictedset=indictedset
                                   +[suspected(*hsw1.mrar*)];
               0(*write*):if lastbususer=cpuid then settemptrans
                          else indictedset:=indictedset
                                     +[ppusoftname(*lastbususer*)]
            end(*case*)
        end

  (*turn the peripheral units and mmu off*)

        if (lastbususer=dma1id or lastbususer=dma2id or lastbususer=siuid)
           then lastusedmmm:=mrusedmmm;
        for i:=1 to mmusize do
           poweroff(mmuid,i);
        for i:=1 to 2 do
           begin
              poweroff(dma1id,i);
              poweroff(dma2id,i);
              poweroff(siuid,i)
            end;

  (*cat4d test*)
        if faultcat=cat4d then
        begin
           if lastbususer=cpuid then
              indictedset:=indictedset+[suspected]
           else begin
              indictedset:=indictedset+[suspectd];
              indictedset:=indictedset+[mrusermmm]
           end
        end;

  (*find the system memory modules *)
        findsysmodid(numfound, sysmodid);

  (*system map recovery*)
        case numfound of

        0:(*no system memory module found*)
           begin
              coldrestart;
              go to 1000 (*end of program*)
           end;
        1:(*only one system memory module  found*)
           begin
              poweron(mmuid,sysmodid[0]);
              if mmu_hardval(0,sysmodid[0]) then
                 begin
                    status[id(mmuid,sysmodid[0])]:=working;
                    makecopy(0,sysmodid[0],1,sysmodid[1]);
                      (*using the memory module with soft  name  0  and
                      hard  name  sysmodid[0]  create  the other system
                      memory module.  assign the  soft  name  of  newly
                      made system memory module to  1  and  assign  the
                      hardname of the module to sysmodid[1]*)
                    record(1,sysmodid[1]);
                    constlost[1]:=yes;
                    status[id(mmuid,sysmodid[1])]:=working
                 end
              else
```

```
          if refreshable(0,sysmodid[0]) then
            (*is the module with soft name 0 and hard name
             sysmod[0] refreshable ??*)
          begin
             refresh(0,sysmodid[0]);
                    (*refresh the module with soft
                    name 0 and hard name sysmodid[0]*)
             status[id(mmuid,sysmod[0])]:=working;
             makecopy(0,sysmodid[0],1,sysmodid[1]);
             record(0,sysmodid[0]);
             record(1,sysmodid[1]);
             constlost[1]:=yes;
             status[id(mmuid,sysmodid[1])] :=working
          end
          else begin
             poweroff(mmuid,sysmodid[0])
             coldrestart;
             go to 1000 (*end of program*)
          end
    end;
 2:(*both system memory modules are found*)
    begin
       poweron(mmuid,sysmodid[0]);
       if mmu_hardval(0,sysmodid[0]) then
          status[id(mmuid,sysmodid[0])]:=working
       else poweroff(mmuid,sysmodid[0]);
       poweron(mmuid,sysmodid[1]);
       if mmu_hardval(1,sysmodid[1]) then
          status[id(mmuid,sysmodid[1])]:=working
       else poweroff(mmuid,sysmodid[1]);
       case status[id(mmuid,sysmodid[0])] of
       working:
          case status[id(mmuid,sysmodid[1])] of
          working:;(*perfect*)
          faulty :begin
                     rerconfsysmod(0,1);
                     refresh(1,sysmodid[1]);
                     record(1,sysmodid[1]);
                     status[id(mmuid,sysmodid[1])]:=working
                  end
          end(*case*)
       faulty:
          case status[id(mmuid,sysmodid[1])] of
          working:reconfsysmod(1,0);
          faulty :
             begin
                poweron(mmuid,sysmodid[0]);
                if refreshable(0,sysmodid[0]) then
                   refreconf(0,1)
                else begin
                   poweroff(mmuid,sysmodid[0]);
                   poweron(mmuid,sysmodid[1]);
                   if refreshable(1,sysmodid[1]) then
                      refreconf(1,0)
                   else begin coldrestart; go to 1000 end
```

```
                    end
                 end
              end(*case*)
           end(*case*)
        end
     end(*case*);

(* store the proper soft names of system memory modules into the
corresponding soft name registers*)

     case mmmsoftname(sysmodid[0]) of
     0: begin
           mmu[sysmodid[0]].soft1=0;
           mmu[sysmodid[0]].soft2=0;
           mmu[sysmodid[1]].soft1=1;
           mmu[sysmodid[1]].soft2=1
        end;
     1: begin
           mmu[sysmodid[0]].soft1=1;
           mmu[sysmodid[0]].soft2=1;
           mmu[sysmodid[1]].soft1=0;
           mmu[sysmodid[1]].soft2=0
           i:=constlost[1];
           constlost[1]:=constlost[0];
           constlost[0]:=i;
        end
     end(*case*);

(*dma1 test*)
     with mmu[sysmodid[0]].sysmodcontent.ppmask do
     begin
        if dma1mask[26]=0 and dma1mask[27]=0 then
            (*this cannot occur under norma' circumstances*)
        begin coldrestart; go to 1000  end;
        if dma1mask[26] <> dma1mask[27] then
        begin
           if faultcat=cat4a or 16 in indictedset then
              pp_test(dma1id,(*hardval:=*)yes)
           else pp_test(dma1id,(*hardval:=*)no)
        end
     end;

(*dma2 test*)
     with mmu[sysmodid[0]].sysmodcontent.ppmask do
     begin
        if dma2mask[28]=0 and dma2mask[29]=0 then
        begin coldrestart; go to 1000  end
        if dma2mask[28] <> dma2mask[29] then
        begin
           if faultcat=cat4b or 17 in indictedset then
              pp_test(dma2id,(*hardval:=*)yes)
          else pp_test(dma2id,(*hardval:=*)no)
       end
    end;
```

```
(*siu test*)
      with mmu[sysmodid[0]].sysmodcontent.ppmask do
      begin
         if siumask[30]=0 and siumask[31]=0 then
            begin coldrestart; go to 1000 end;
         if siumask[30 <> siumask[31] then
         begin
            if faultcat=cat4c or 18 in indictedset then
               pp_test(siuid,(*hardval:=*)yes)
            else pp_test(siuid,(*hardval:=*)no)
         end
      end

(*recovery of the application memory modules*)

      for i:=2 to 14 do (*soft name*)
      begin
         j:=mmmhardname(i);
         poweron(mmuid,j);
         mmu[j].soft1:=i;
         mmu[j].soft2:=i;
         if i in indictedset then
         begin
            if mmu_hardval(i,j) then
            begin
               status[id(mmuid,j)]:=working;
               setcat4trans(mmuid,j)
            end
         end
         else
            if mmu_weakval(i,j) then
               status[id(mmuid,j)]:=working;
         if status[id(mmuid,j)]=faulty then
         begin
            if refreshable(i,j) then
            begin
               refreshamm(i,j,k);(*if there is no spare module which
                                  can be assigned to softname i, then
                                  k is set to 'no' on return*)
               if k=yes then
               begin
                  record(i,j);
                  status[id(mmuid,j)]:=working
               end
               else
                  for k:=0 to 1 do
                     mmu[sysmodid[k]].sysmodcontent.softtable(i).lost
                        :=1
            end
            else
               for k:=0 to 1 do
                  mmu[sysmodid[k]].sysmodcontent.softtable(i).lost:=1
         end
      end;
      recover_systmemconst; (*recover system memory constant info
                                   that has not been recovered*)
        recover_applmemlost;       (*recover application memory modules that
                                   have not been recovered*)

    (*update transient fault  count*)
        updatetranscount;

      1000:
      end(*primary*)

      else-by

    (*----------------------------*)
    (*alternate recovery procedure*)
    (*----------------------------*)

      begin

    (*validation of cpu's capability of reading hsw1*)
        for i:=5 to 8 do
           if hsw1.fault_cat[i]=1 then pflag6; pflag7 end;

    (*initialization*)
        for i:=0 to lastmodid do status[i]:=faulty;
        for i:=0 to 14 do constlost[i]:=no;

    (*cpu test*)
        if cpu_hardval then
        begin
           status[id(cpuid,acpuid)]:=working;
           status[id(cpuid,mcpuid)]:=working
        end;

    (*bus test*)
        reconfcount:=0;
        while not bus_val do
        begin
           reconfcount:=reconfcount+1;
           if reconfcount>23 then cycle_cpu
           else begin
              pflag6;
              cycle_abus;
              cycle_dbus;
              pflag7
           end(*else*)
        end;
        status[id(abusid,abusspid)]:=working;
        status[id(dbusid,dbusspid)]:=working;

    (*turn the peripheral units and mmu off*)
        for i:=1 to mmusize do
           poweroff(mmuid,i);
        for i:=1 to 2 do
        begin
           poweroff(dma1id,i);
```

```
          poweroff(dma2id,i);
          poweroff(siuid,i)
        end;

  (*recovery of peripheral units and mmu*)
        coldrestart;

     end(*alternate*)

     else-error;

     start_normal_processing

  end(*program*)
```

Appendix C: Module test procedures

Test procedures for several modules are detailed in this appendix.

C.1 CPU module test

Due to a large number of functional sections within a CPU module, a spectrum of tests can be designed according to the degree of coverage desired. To the least extent, a weak test of CPU should perform the following tasks:
(1) Test of the monitoring capability of the M-CPU: the bits from position 24 to position 31 in the M-CPU mask register are set to cause miscomparison of control signal lines.
(2) Exercise of the ALU and registers (by executing a routine): The result of this routine should cause some predetermined program flags to be generated and sent to the CCU for the comparison purpose.
(3) Bus code monitor test: pp.3-145 of [S1] (an improperly coded information is established on the bus using the "store with bad parity data/address" instruction; a fault interrupt should be generated if the bus code monitor detects the invalid code).

A strong CPU test can perform more extensive exercises of various functional sections including the address/data comparator, the control comparator, etc.

C.2 peripheral module test

Since the functions of the DMA units and the SIU are similar, i.e. to serve as the focal point of communication between the computer and the surrounding environment, only the DMA module tests will be presented. A weak test of a DMA module can be done simply by sending a dummy command (using control word 1) to the DMA module and then observing if this command is properly received by the module. A strong test can perform the following:

(1) Weak test.

(2) Watch dog timer test:

(2.1) primary: pp.3-148 of [S1].

(2.2) alternate 1: the CPU sends a reconfiguration ROM address to the DMA and command the DMA to request data transfer from this address. No memory module will respond.

(2.3) alternate 2: arm one memory module with ignore write and cause the DMA to send data to this memory module.

(3) Bus code monitor test: pp.3-145 of [S1].

C.3 MMU module test

Each memory module must be at least checked for its capability to read and write and the correctness of its rippler registers. Thus a weak test should perform:

(1) Test of the memory read/write capability: a word is read from an arbitrary location in the memory into a CPU register and then written back into the same location.

(2) Verification of the rippler register content against the information stored in the system map.

A strong memory test can perform the following:

(1) Weak test.

(2) Write protect monitor test:

(2.1) primary: pp.3-146 of [S1].

(2.2) alternate: for each of the four protected regions attempt to read a word and then write back to the same location. If any of the blocks is protected, a write protect violation would have been generated. If not, set write protect for each of the four blocks in sequence and then do read and write operations for each block to cause an interrupt.

(3) Soft name monitor test:

(3.1) primary: pp.3-146 of [S1].

(3.2) alternate: use memory monitor test functions CSSU,ZCON, and ZCOFF to validate this monitor.

(4) Syndrome monitor test:

(4.1) primary: pp.3-146 of [S1].

(4.2) alternate:

a) reconfigure bit lines and store two calculated words in any locations.

b) go back to the original bit line configuration and read the first word. This should generate a fault and cause syndrome monitor register to be filled.

c) enter refresh mode and read the second word.

d) the second word will generate another syndrome different from the first one.

e) syndrome fault will then be indicated.

Note: these two words should be so prepared that each will cause one bit error in different positions when the original bit line configuration is used.

(5) Bus code monitor test: pp.3-145 of [S1].

(6) Analog monitor test: pp.3-146 if [S1].

(7) Data decoder test: pp.3-145 of [S1].

(8) Address decoder test: pp.3-146 of [S1].

Part II

A TRANSLATOR OF PASCAL AUGMENTED WITH RECOVERY BLOCK

by

K. H. Kim and T. Arshi

Abstract: Recovery block is a language construct designed to support structured incorporation of program redundancy. In order to facilitate a study on the use of design/program redundancy, often called fault-tolerant programming, a software that translates a program written in PASCAL augmented with the recovery block into an equivalent program in ordinary PASCAL, was developed. The translator itself is written in PASCAL. The translation strategy and the organization of the translator are explained. The complete listing of the translator is given together with some test-run outputs.

1. Introduction

Recovery block is a language construct devised to support redundant design, also called *fault-tolerant programming*, an approach to obtaining a software tolerant of residual design errors/ inadequacies [H1]. In order to experiment with redundant design, the recovery block was introduced into PASCAL and the result is called a *fault-tolerant (FT-) PASCAL* in this paper. It was then necessary to develop a translator of FT-PASCAL so that the programs written with recovery blocks could be executed.

There are two approaches to translation of FT-PASCAL programs. One is to translate the programs directly into machine programs, and the other is to first translate the programs into ordinary PASCAL programs and then translate the PASCAL programs into machine programs by using already available PASCAL compilers. The former approach has an advantage of producing more efficient machine programs. On the other hand, the latter approach has an advantage of being easily transportable. That is, once a preprocessor which translates FT-PASCAL programs into ordinary PASCAL programs and is a PASCAL program itself, is developed, then the preprocessor can be transported to any installation equipped with a PASCAL compiler and can support fault-tolerant programming.

In this paper a FT-PASCAL preprocessor is described. The pr~~o~~eprocessor is written in PASCAL and has been operational since May, 1978. In the next section, several FT-PASCAL programs are shown together with functionally equivalent (ordinary) PASCAL programs. The organization of the preprocessor is described in section 3 and some possible extensions are mentioned in section 4. The complete listing of the preprocessor is given in Appendix A and some test-run results are given in Appendix B.

2. Translation of FT-PASCAL into PASCAL

A PASCAL procedure is recursively defined as a sequence of constant declarations, variable declarations, (nested) procedures/functions, and the body which may be a simple or compound statement. All the components except the body may be empty. An entire PASCAL program can be viewed as a (main) procedure. The FT-PASCAL allows recovery blocks to be included in the procedure bodies. The control flow implied by a recovery block can be explicitly expressed in PASCAL by using conditional statements. For example, see Figure 1.

Basically a recovery block is equivalent to the following PASCAL construct.

```
     <save: save variables that may be modified inside the
            recovery block>
     <primary object block 0.1>
     if <acceptance test> then (*exit*) goto 100;
     <restore: restore the variables that could have been
            modified inside the recovery block>
     <alternate object block 0.2>
     if <acceptance test> then (*exit*) goto 100;
     <restore>
     <0.3>
     -------
     -------
     if <acceptance test> then (*exit*) goto 100;
     <print-error: print error message>
100:<purge: purge the saved variable values>
```

```
         ⋮
  ensure     |$X^2$ - prior (X)| < ∈

    by         X := SQRTA (X)

    else-by    X := SQRTB (X)

  else-error
         ⋮
```

Figure 1a. An FT-PASCAL program

```
         ⋮

< Saving of Non-local Variables To Be Changed >

    X := SQRTA (X) ;

  if |$X^2$ - XS| < ∈ then goto CONTINUE ;

< Restoration of Changed Non-local Variables >

    X := SQRTB (X) ;

  if |$X^2$ - XS| < ∈ then goto CONTINUE ;

< Send Error Message >

CONTINUE : < Purge Saved Variable Values >
         ⋮
```

Figure 1b. An ordinary PASCAL program equivalent to the program in 1a.

The first component in the above PASCAL construct is responsible for saving the variables that may be changed inside the recovery block. Since the exact set of variables that are modified varies from execution to execution, the preprocessor must save all the variables that appear as destinations of assignment statements or actual parameters of procedure/function calls. A simple approach adopted is to declare a new variable, called a backup variable, for each of those variables (which may be called main variables to distinguish from backup variables), and then to save the content of each main variable into its backup variable. Each variable and its backup variable must be of the same type.

Since variables cannot be declared within a procedure body, the above constructs need to be implemented as a procedure. If the acceptance test fails, then the main variables are restored to their original values kept in their backup variables. Note that the specifications of acceptance test and restore operation appear repetitively in the above description; it is thus useful to implement those operations as two nested procedures that are repetitively called. The purge operation in the above description is equivalent to discarding the backup variables. If the above construct is implemented into a procedure, the purge is automatic on exit from the procedure. Therefore, a recovery block can be translated into the following PASCAL procedure, called a <u>fault-tolerant (FT-) procedure</u>, and a calling statement.

```
procedure FT;
   var <backup-variable declarations>
   procedure SAVE;
       <save the contents of main variables into backup
        variables>
   procedure RESTORE;
       <restore main variables by using the contents of
```

```
                backup variables>
            function ACCEPTABLE;
                <acceptance test>
        begin
            SAVE;
            <0.1>;
            if ACCEPTABLE then goto 100;
            RESTORE;
            <0.2>;
            if ACCEPTABLE then goto 100
            -------
            -------
            if ACCEPTABLE then goto 100
            <print error message>
    100:end
```

3. Organization of the preprocessor

The preprocessor goes through one "analysis" pass of the source program, n "translation" passes where n is the maximum level of nesting of recovery blocks, and a "merge" pass. In pass 1 the preprocessor produces a compact representation of the information on program structure and vocabulary. The set of variables that may be changed inside each recovery block are also recognized during this pass. In pass 2 (i.e., first translation pass), the first-level recovery blocks (i.e., recovery blocks not nested within other recovery blocks) are translated into FT-procedures and stored together into a new file called text-increment-1 while procedure calls replace the recovery blocks in the source file. The second-level recovery blocks nested within the first-level recovery blocks are not translated at this time. The second-level recovery blocks (in file text-increment-1) are translated in pass 3 and newly generated FT-procedures are stored into a new file text-increment-2. Thus each translation pass processes only the outer-most recovery blocks contained in the text-increment file produced through the preceding pass. After all the recovery blocks are translated, the main source file and all the text-increment files containing FT-procedures are merged. This unoptimized simplistic approach is motivated by the simplicity in logic and the prevention of the source text file from growing during translation.

The following words are reserved for use by the preprocessor and thus should not be used in source programs.

(1) Any character string of which the first 8 characters are "RB/SA/RS/VT/BV/TP" followed by exactly 6 digits, where "/" denotes "or".

(2) FAULTFLAG.

4. Extensions

The recovery block is not the only high level language construct that is useful in fault-tolerant programming. Horning et al [H1] proposed a recoverable procedure that allows the incorporation of a special procedure of restoring computation, i.e., a procedure different from undoing variable assignments. Some other possible extensions of the recovery block are also discussed in [K1]. The current preprocessor does not support any of these recovery block extensions.

As shown in the outputs of test-runs in Appendix B, the preprocessor attempts to indent the output (i.e., PASCAL programs) to a limited extent. Although the PASCAL programs generated may be read infrequently, incorporation of a more thorough indentation capability into the preprocessor seems to be a worthwhile extension.

Reference

[H1] Horning, J.J. et al, "A program structure for error detection and recovery", Lecture Notes in Comp. Sci., vol. 16, Springer-Verlag, 1974, pp.171-187.

[K1] Kim, K.H. and Ramamoorthy, C.V., "Recent developments in software fault tolerance through program redundancy", Proc. 10th Hawaii Int'l Conf. on System Sciences, Jan. 1977, pp.234-238.

Appendix A.

FT-PASCAL

Preprocessor

```
(***********************************************************************)
(*                                                                     *)
(*                                                                     *)
(*                                                                     *)
(*                                                                     *)
(*                                                                     *)
(*                                                                     *)
(*  THE PURPOSE OF THIS PREPROCESSOR IS TO ACCEPT A PASCAL            *)
(*  PROGRAM AND TRANSLATE ALL ITS RECOVERY BLOCKS INTO                 *)
(*  PROCEDURES SUCH THAT THE OUTPUT OF PREPROCESSOR BE                 *)
(*  ACCEPTED BY CONVENTIONAL PASCAL COMPILER.                          *)
(*  GENERALLY IT WORKS AS FOLLOWS:                                     *)
(*                                                                     *)
(*                                                                     *)
(*     PROGRAM           I--------------I        I----------I          *)
(*    INCLUDING ---> I    PRE.      I ---> I  PASCAL  I --->           *)
(*    REC.BLOCK      I  PROCESSOR   I      I COMPILER I                *)
(*                   I              I      I          I                *)
(*                   I______________I      I__________I                *)
(*                                                                     *)
(*  THE PREPROCESSOR CONSISTS OF TWO PASSES. PASS ONE SCANS            *)
(*  THRU THE INPUT PROGRAM AND COLLECTS SOME INFORMATION               *)
(*  INTO DIFFERENT TABLES USED IN PASS TWO. IN FACT PASS_1             *)
(*  BUILDS THE FOLLOWINGS:                                             *)
(*                                                                     *)
(*     1.  PROCEDURE/RECOVERY BLOCK TREE WHERE MAIN PROGRAM            *)
(*         IS TREATED AS THE ROOT OF THE TREE.                         *)
(*     2.  VARIABLE DECLARATION TREE,WHICH ENABLE PASS_2 TO            *)
(*         RECOGNIZE GLOBAL AND LOCAL VARIABLES.                       *)
(*                                                                     *)
(*  PASS_2 GOES THRU THE INPUT PROGRAM AGAIN, USING THE                *)
(*  COLLECTED INFORMATION BY PASS_1, TRANSLATES FIRST LEVEL            *)
(*  RECOVERY BLOCKS AND LEAVES THE SECOND,THIRD,... LEVEL              *)
(*  REC.BLOCKS AS THEY ARE. PASS_2 IS CALLED AS LONG AS THERE          *)
(*  ARE MORE RECOVERY BLOCKS TO BE TRANSLATED.                         *)
(*                                                                     *)
(*     SPECIAL NOTES                                                   *)
(*     =============                                                   *)
(*                                                                     *)
(*     1.  NONE OF THE FOLLOWING NOTES ARE RESTRICTIONS TO             *)
(*         THE USER. BY CHANGING A ' CONST ' CORRESPONDING TO          *)
(*         EACH, FLEXIBILITY IS OBTAINED.                              *)
(*     2.  IDENTIFIER LENGTH IS CONSIDERED TO BE 8. MORE THAN          *)
(*         THAT WILL BE TRUNCATED.(CAN CHANGE THE ' CONST ').          *)
(*     3.  LINE LENGTH IS CONSIDERED TO BE 80. FOR MORE LENGTH         *)
(*         SHOULD CHANGE THE RELATED ' CONST ' IN PROGRAM.             *)
(*     4.  PROGRAM LENGTH HAS NO RESTRICTION AT ALL.                   *)
(*                                                                     *)
(*                                                                     *)
(***********************************************************************)


CONST TEXT_LENGTH=100;
      NO_OF_PROC=20;
      VD_TBL_LENGTH=100;
      PTRS_LENGTH=82;
      HASH_PTRS_LENGTH=37;
      LINE_LENGTH=80;
      ID_LENGTH=8;
      ID_LENG_2=16;
      NO_NESTED_WITH=10;
      TYPE_LENGTH=40;
      NO_OF_COMP=50;
      NO_OF_SIMPLE=50;
      NL_VAR_TBL_LENGTH=100;
TYPE  IDTYPE=PACKED ARRAY[1..TYPE_LENGTH] OF CHAR;
      TEXT_FILE=FILE OF CHAR;
      TEXT_LINE=ARRAY[1..LINE_LENGTH] OF CHAR;
      STRING47=PACKED ARRAY[1..47] OF CHAR;
      STRING45=PACKED ARRAY[1..45] OF CHAR;
      STRING30=PACKED ARRAY[1..30] OF CHAR;
      STRING27=PACKED ARRAY[1..27] OF CHAR;
      STRING16=PACKED ARRAY[1..16] OF CHAR;
      STRING10=PACKED ARRAY[1..10] OF CHAR;
      STRING9 =PACKED ARRAY[1..9]  OF CHAR;
      STRING20=PACKED ARRAY[1..20] OF CHAR;
      STRING6=PACKED ARRAY[1..6] OF CHAR;
      STRING5=PACKED ARRAY[1..5] OF CHAR;
      STRING4=PACKED ARRAY[1..4] OF CHAR;
      STRING3=PACKED ARRAY[1..3] OF CHAR;
      STRING2=PACKED ARRAY[1..2] OF CHAR;
      IDENTIFIER=PACKED ARRAY[1..ID_LENGTH] OF CHAR;
      DOUBLE_ID =PACKED ARRAY[1..ID_LENG_2] OF CHAR;
      LINKAGE_PLACES=RECORD
                         POS:INTEGER;
                         NAME:IDENTIFIER
                     END;
      VD_INFO=RECORD
                  VNAME:IDENTIFIER;
                  TCODE:INTEGER;
                  TTCODE:INTEGER;
                  INDIC:INTEGER
          END;
      VD_NODE=RECORD
                  PTR_TO_VDTABLE:INTEGER;
                  SIZE:INTEGER;
                  PR_NAME:IDENTIFIER;
                  FATHER:INTEGER;
                  FIRST_SON:INTEGER;
                  YOUNGER_BROTHER:INTEGER
          END;
      HASHED_PNAME=RECORD
```

```
                                PTR_TO_PROC_TREE:INTEGER;
                                SEQ_ADR:INTEGER
                          END;
      COMPONENT=RECORD
                             LEVEL:INTEGER;
                             NAME:IDENTIFIER;
                             IND1:CHAR;
                             IND2:CHAR;
                             TCODE:INTEGER
                      END;
      PROC_NODE=RECORD
                             PR_RB_NAME:IDENTIFIER;
                             INDICATOR:STRING2;
                             PTR_TO_VDNODE:INTEGER;
                             PTR_TO_END_DCL:INTEGER;
                             NO_NLVAR_USED:INTEGER;
                             PTR_TO_NLVAR:INTEGER;
                             FATHER:INTEGER;
                             FIRST_SON:INTEGER;
                             ENTRY:INTEGER;
                             LABLE:INTEGER;
                             MARK:CHAR;
                             YOUNGER_BROTHER:INTEGER
                      END;
      NL_VAR=RECORD
                          INDICATOR:STRING2;
                          PTR_TO_VDTAB:INTEGER
                 END;
      PROC_INFO=RECORD
                             INDICATOR:STRING2;
                             NAME:IDENTIFIER
                      END;
VAR   TEXT1:FILE OF CHAR;
      TEXT2:FILE OF CHAR;
      TEXT3,TEXT4,INTX4,INTX5:TEXT_FILE;
      TEMP_STMT:ARRAY[1..LINE_LENGTH] OF CHAR;
      LINKAGE:ARRAY[1..LINE_LENGTH] OF CHAR;
      BUFER:ARRAY[1..LINE_LENGTH] OF CHAR;
      NEW_TYPE:ARRAY[1..LINE_LENGTH] OF CHAR;
      NO_NEW_TYPE,POS_NEW_TYPE:INTEGER;
      INDIC,CNK:INTEGER;
      TIPE,TI:IOTYPE;
      TYP:IDENTIFIER;
      TOKEN,PREV_TOKEN:STRING20;
      PROC_NAME:IDENTIFIER;
      TXT_PTR , STMT_NO , LINE_NO:INTEGER;
      SYMBOLS , DELIMETERS:SET OF CHAR;
      PROC_RB_TREE:ARRAY[1..NO_OF_PROC] OF PROC_NODE;
      NEXT_PROC_NODE:INTEGER;
      STACK:ARRAY[1..NO_OF_PROC] OF PROC_INFO;


      TOP_OF_STACK:INTEGER;
      HASH_TABLE:ARRAY[1..NO_OF_PROC] OF HASHED_PNAME;
      HASH_PTRS:ARRAY[0..HASH_PTRS_LENGTH] OF INTEGER;
      HASH_INDEX,HI:INTEGER;
      VAR_DCL_TREE:ARRAY[1..NO_OF_PROC] OF VD_NODE;
      VAR_DCL_TABLE:ARRAY[1..VD_TBL_LENGTH] OF VD_INFO;
      STRUCTURE_TABLE:ARRAY[1..NO_OF_COMP] OF COMPONENT;
      SIMPLE_TABLE:ARRAY[1..NO_OF_SIMPLE] OF IDENTIFIER;
      TYPSW,STT,STRT,ASFR:INTEGER;
      LAST_PROC:IDENTIFIER;
      NEXT_VD_NODE ,NEXT_VD_ROW:INTEGER;
      NL_VAR_TABLE:ARRAY[1..NL_VAR_TBL_LENGTH] OF NL_VAR;
      NLVAR_LIST:ARRAY[1..NL_VAR_TBL_LENGTH] OF INTEGER;
      NO_NLOC:INTEGER;
      NL_BUFER:ARRAY[1..NL_VAR_TBL_LENGTH] OF NL_VAR;
      NEXT_NL:INTEGER;
      WITH_STACK:ARRAY[1..NO_NESTED_WITH,1..2] OF INTEGER;
      WITH_SW,TWS:INTEGER;
      NEXT_NL_VAR:INTEGER;
      RB_NAME,TP_NAME:IDENTIFIER;
      CM:IDENTIFIER;
      PTR1,PTR2,PTR3,CODE,LEV:INTEGER;
      PRB,IDT:STRING2;
      FATHER_NAME:IDENTIFIER;
      PROC_CALL,FIRST_CALL,MORE_VAR,ASSIGNED_VAR,READ_ASSIGNED:BOOLEAN;
      DOT,SEMI_COLON:BOOLEAN;
      ID:IDENITFIER;
      ST:STRING3;
      FL:STRING4;
      AR:STRING5;
      RC:STRING6;
      CH:CHAR;
      WC,LASTNODE:INTEGER;
      STK:ARRAY[1..5] OF INTEGER;
      TS,ERR_CODE:INTEGER;
      NEED_MERGE,NEED_MORE_PASS:BOOLEAN;
      DCL_TYPE,SEMICOLON:BOOLEAN;
      LASTELSERROR,LEE:INTEGER;
      COL,LIM,PVDT,GOTOLAB,SW:INTEGER;
      PVS,PRV,FNT:IDENTIFIER;
      COLMAKER:INTEGER;

PROCEDURE GIVE_TYPE(VAR TYP:IDENTIFIER; VAR POTR:INTEGER);
    (*  TO GET THE TYPE OF A COMPONENT OF SOME  *)
    (*  STRUCTURE VARIABLE THIS PROCEDURE MAY   *)
    (*  BE CALLED GIVING IT A POINTER TO THE    *)
    (*  STRUCTURE TABLE. IT MAY NOT BE USED FOR *)
    (*  SIMPLE VARIABLES TYPE.                  *)

    VAR L,ILO:INTEGER;
    BEGIN ILO:=POTR;
```

```
              IF((STRUCTURE_TABLE[ILO].IND1=' ') AND
                (STRUCTURE_TABLE[ILO].IND2=' '))
               THEN TYP:=STRUCTURE_TABLE[ILO].NAME
              ELSE IF STRUCTURE_TABLE[ILO].IND2='S'
                  THEN BEGIN
                          L:=STRUCTURE_TABLE[ILO].TCODE;
                          TYP:=SIMPLE_TABLE[L];
                       END
                 ELSE IF STRUCTURE_TABLE[ILO].IND1='A'
                     THEN TYP:=STRUCTURE_TABLE[ILO+1].NAME
                     ELSE BEGIN
                             L:=STRUCTURE_TABLE[ILO].TCODE;
                             GIVE_TYPE(TYP,L);
                          END;
         END;

      (****************************************************)
      (*                     PASS ONE                      *)
      (*                                                   *)
      (*   IN GENERAL PASS_1 PERFORMS THE FOLLOWING:       *)
      (*                                                   *)
      (*        INPUT PROGRAM                              *)
      (*             I                                     *)
      (*             I                                     *)
      (*             I                                     *)
      (*                                                   *)
      (*        I-----------I                              *)
      (*        I  PASS_1   I                              *)
      (*        I           I                              *)
      (*        I           I                              *)
      (*        I-----------I                              *)
      (*             I                                     *)
      (*             I                                     *)
      (*             I                                     *)
      (*                                                   *)
      (*        1. PROC/RB TREE:                           *)
      (*           CORRESPONDING TO EACH REC.BLOCK OR      *)
      (*           WILL BE A NODE IN THIS TREE. MAIN       *)
      (*           PROGRAM SITS AS THE ROOT. DETAILS OF    *)
      (*           EACH NODE CAN BE FOUND IN TYPE DCL AREA *)
      (*                                                   *)
      (*        2. VAR DCL TREE:                           *)
      (*           CORRESPONDING TO EACH PROCEDURE WILL BE *)
      (*           A NODE IN THIS TREE,CONTAINING VARIABLE *)
      (*           DECLARATION AREA FOR THAT PROCEDURE.    *)
      (*           BY A POINTER TO VAR_DCL_TABLE ALL THE   *)
      (*           VARIABLES MAY BE FOUND.                 *)
      (*                                                   *)
      (*        3. VAR_DCL_TABLE:                          *)
      (*           A PORTION OF THIS TABLE CONTAINS ALL    *)
      (*           VARIABLES DECLARED IN THE PROCEDURE     *)
      (*           ASSIGNED TO THAT PORTION.               *)

      (*                                                   *)
      (*        4. STRUCTURE_TABLE:                        *)
      (*           CONTAINS ALL STRUCTURE VARIABLES EITHER *)
      (*           DECLARED AS A TYPE OR VAR. NESTED       *)
      (*           RECORDS ARE SHOWN BY LEVEL NUMBER. FOR  *)
      (*           MORE DETAIL REFER TO TYPE DCL AREA.     *)
      (*                                                   *)
      (*        5. OTHER TABLES SUCH AS STACK,HASH_TABLE,  *)
      (*           SIMPLE_TABLE,ETC.                       *)
      (*                                                   *)
      (*   BESIDES, PASS_1 INSERTS A '$LINKAGE' STATEMENT  *)
      (*   WHERE LATER ON, SOMETHING IS GOING TO BE        *)
      (*   INSERTED INTO THE TEXT.                         *)
      (*                                                   *)
      (****************************************************)

      PROCEDURE PASS_1;

         PROCEDURE INITIALIZE_PASS1;
            BEGIN
               STMT_NO:=0; TOP_OF_STACK:=1; NEXT_NL_VAR:=1; LINE_NO:=0;
               NEXT_VO_NODE:=1; NEXT_VO_ROW:=1; NEXT_PROC_NODE:=1;
               DELIMETERS:=['+','-','*','/','>','<','=','^',
                            ',',';',':','(',')','[',']',' '];
               SYMBOLS:=['A'..'Z','0'..'9','_','$'];
               INDIC:=0;   HASH_INDEX:=1;
               NO_NEW_TYPE:=0;    POS_NEW_TYPE:=-1;
               RB_NAME:='RB000000';   TP_NAME:='TP000000';
               FOR HI:=0 TO HASH_PTRS_LENGTH DO  HASH_PTRS[HI]:=0;
               FIRST_CALL:=TRUE; READ_ASSIGNED:=FALSE; PROC_CALL:=FALSE;
               REWRITE(TEXT1);   REWRITE(TEXT2);
               TWS:=1;  COLMAKER:=0;
               PVS:='SA000000'; PRV:='PS000000'; FNT:='VT000000';
               DCL_TYPE:=TRUE;    SEMICOLON:=TRUE;
               GOTOLAB:=1357;
               TYPSW:=2;  STT:=1;  STRY:=1;
               NEXT_NL:=1;
            END;

         PROCEDURE PRINT_RESULT;
            VAR I:INTEGER;
            BEGIN
               WRITELN;  WRITELN;
               WRITELN(' .... PROCEDURE AND RECOVERY BLOCK TREE ....');
               WRITELN;WRITELN;
               WRITELN(' PROC NAME',' INDICATOR',' PTR TO VO TREE',
                     ' END OF DCL',' NL VAR USED',' PTR TO NLVAR',
                       ' FATHER',' SON',' BROTHER',' ENTRY',' LABEL');
               WRITELN;
               FOR I:=1 TO NEXT_PROC_NODE-1 DO
      WRITELN(PROC_RH_TREE[I].PR_RH_NAME,'          ',
             PROC_RH_TREE[I].INDICATOR,
             PROC_RH_TREE[I].PTR_TO_VDNCDE:11,
             PROC_RH_TREE[I].PTR_TO_END_DCL:15,
             PROC_RH_TREE[I].NO_NLVAR_USED:11,
             PROC_RH_TREE[I].PTR_TO_NLVAR:14,
             PROC_RH_TREE[I].FATHER:11,
             PROC_RH_TREE[I].FIRST_SON:7,
             PROC_RH_TREE[I].YOUNGER_BROTHER:6,
             PROC_RH_TREE[I].ENTRY:10,
             PROC_RH_TREE[I].LABLE:7);
      WRITELN;WRITELN;
      WRITELN(' .... VAR DCL TREE ....'); WRITELN; WRITELN;
      WRITELN('     PTR TO VDTABLE','        SIZE','         PROC NAME',
      '     FATHER','     FIRST SON','     BROTHER');
      WRITELN;
      FOR I:=1 TO NEXT_VD_NODE-1 DO
      WRITELN(VAR_DCL_TREE[I].PTR_TO_VDTABLE,
             VAR_DCL_TREE[I].SIZE:15,'          ',
             VAR_DCL_TREE[I].PR_NAME,
             VAR_DCL_TREE[I].FATHER:8,
             VAR_DCL_TREE[I].FIRST_SON:14,
             VAR_DCL_TREE[I].YOUNGER_BROTHER:14);
      WRITELN;  WRITELN;
      WRITELN(' .... VAR DCL TABLE ....'); WRITELN; WRITELN;
      WRITELN(' TCODE',' NAME','     TTCODE','   INDIC');
      WRITELN;
      FOR I:=1 TO NEXT_VD_ROW-1 DO
      WRITELN(VAR_DCL_TABLE[I].TCODE:4,'  ',
             VAR_DCL_TABLE[I].VNAME,'  ',
             VAR_DCL_TABLE[I].TTCODE:3,
             VAR_DCL_TABLE[I].INDIC);
      WRITELN;WRITELN;
      WRITELN(' .... HASH TABLE ....'); WRITELN;  WRITELN;
      WRITELN(' PROC NAME','  PTR TO PROC TREE','  SEQ ADR');WRITELN;
      FOR I:=1 TO HASH_INDEX-1 DO
      WRITELN(HASH_TABLE[I].PR_NAME,
             HASH_TABLE[I].PTR_TO_PROC_TREE,
             HASH_TABLE[I].SEQ_ADR);
      WRITELN;WRITELN;
      WRITELN(' .... NONLOCAL VAR TABLE ....'); WRITELN;WRITELN;
      WRITELN(' INDICATOR','  TO VD TABLE');  WRITELN;
      FOR I:=1 TO NEXT_NL_VAR-1 DO
      WRITELN('      ',NL_VAR_TABLE[I].INDICATOR,'    ',
             NL_VAR_TABLE[I].PTR_TO_VDTAB:6);
      WRITELN;WRITELN;
      WRITELN(' .... SIMPLE TABLE .... '); WRITELN; WRITELN;
      FOR I:=1 TO STT-1 DO  WRITELN('        ',SIMPLE_TABLE[I]);
      WRITELN;  WRITELN;
      WRITELN(' .... STRUCTURE TABLE ....'); WRITELN; WRITELN;
      WRITELN('LEVEL','     NAME   ','     ',' TCODE'); WRITELN;


      FOR I:=1 TO STRT-1 DO
      WRITELN(STRUCTURE_TABLE[I].LEVEL:3,'     ',
             STRUCTURE_TABLE[I].NAME,' ',
             STRUCTURE_TABLE[I].IND1,' ',
             STRUCTURE_TABLE[I].IND2,'  ',
             STRUCTURE_TABLE[I].TCODE:5);
      WRITELN;  WRITELN;
      WRITELN(' .... NEW TYPE POSITION=',POS_NEW_TYPE:2);WRITELN;
      WRITELN(' .... NEW TYPE DCL SIZE=',NO_NEW_TYPE:2);
   END;
(*****************************************************)
(*              PROC_DCL_PROCESSOR                  *)
(*                                                  *)
(*  GOES THRU INPUT PROGRAM TOKEN AFTER TOKEN       *)
(*  AND ON HITTING A PROCEDURE WILL DO THE          *)
(*  FOLLOWING:                                      *)
(*                                                  *)
(*   1. FINDS ALL DECLARATIONS AND SAVES THEM       *)
(*   2. GOES THRU THE PROCEDURE BODY AND            *)
(*      FINDS ALL CHANGABLE NONLOCAL VARS           *)
(*      AND STORES THEM IN NL_VAR_TABLE TO BE       *)
(*      USED IN PASS 2.                             *)
(*   3. IN CASE OF NESTED PROCEDURES WILL           *)
(*      CALL ITSELF RECURSIVELY AND DOES THE        *)
(*      SAME THING AGAIN.                           *)
(*                                                  *)
(*  IN GENERAL THIS PROCEDURE CONSISTS OF THE       *)
(*  FOLLOWING PROCEDURES:                           *)
(*                                                  *)
(*   .VAR_DCL_PROCESSOR.....                        *)
(*   .BODY_PROCESSOR........                        *)
(*   .REC_BLOCK_PROCESSOR...                        *)
(*   .WITH_PROCESSOR........                        *)
(*   .O T H E R S...........                        *)
(*                                                  *)
(*****************************************************)

PROCEDURE PROC_DCL_PROCESSOR;

   PROCEDURE GET_TOKEN;
      VAR I,J,P,N:INTEGER;

      PROCEDURE SAVE_STMT;    FORWARD;
      PROCEDURE READ_STMT;
         VAR I,J:INTEGER;
             BLANK,COMENT:BOOLEAN;
         BEGIN
            I:=1;
            IF NOT EOF(INPUT) THEN
            BEGIN
               WHILE NOT EOLN(INPUT) DO
```

THIS PAGE IS BEST QUALITY PRACTICAB
FROM COPY FURNISHED TO DDC

AD-A064 489 UNIVERSITY OF SOUTHERN CALIFORNIA LOS ANGELES F/G 9/2
METHODOLOGIES AND TOOLS FOR DEVELOPING ROBUST FTSC SOFTWARE.(U)
AUG 78 K H KIM F04701-77-C-0120
UNCLASSIFIED SAMSO-TR-78-142 NL

2 OF 2
AD
A0 64489

END
DATE
FILMED
4--79
DDC

```
          BEGIN
             READ(TEMP_STMT[I]);   I:=I+1;
          END;
          READLN;
          FOR J:=I TO LINE_LENGTH DO TEMP_STMT[J]:=' ';
       END;
       STMT_NO:=STMT_NO+1;    TXT_PTR:=1;
       BLANK:=TRUE;  COMENT:=FALSE;
       FOR J:=1 TO LINE_LENGTH-1 DO
       BEGIN    IF TEMP_STMT[J]#' ' THEN
                BEGIN  BLANK:=FALSE;
                       IF ( (TEMP_STMT[J]='(') AND
                            (TEMP_STMT[J+1]='*') ) THEN
                       BEGIN COMENT:=TRUE; GOTO 880; END
                       ELSE GOTO 880;
                END;
       END;
       880:;
       IF (BLANK OR COMENT) THEN
       BEGIN
          INDIC:=1;    SAVE_STMT;  READ_STMT;
       END;
       READ_ASSIGNED:=FALSE;    PROC_CALL:=FALSE;
       INDIC:=1;
    END;

 PROCEDURE SAVE_STMT;
    VAR I,J:INTEGER;
    BEGIN
       IF INDIC=0 THEN BEGIN INDIC:=1; GOTO 70; END;
       IF INDIC=1 THEN BEGIN WRITELN(TEXT1,TEMP_STMT);
                             LINE_NO:=LINE_NO+1; GOTO 70;
                      END;
       I:=1;
       WHILE (TEMP_STMT[I]=' ') DO  I:=I+1;
       FOR J:=1 TO LINE_LENGTH DO
       BEGIN
          LINKAGE[J]:=' ';    BUFER[J]:=' ';
       END;
       LINKAGE[1]:='$';  LINKAGE[2]:='L';  LINKAGE[3]:='I';
       LINKAGE[4]:='N';  LINKAGE[5]:='K';  LINKAGE[6]:='A';
       LINKAGE[7]:='G';  LINKAGE[8]:='E';
       IF (INDIC=2) AND (LNK=4) THEN
BEGIN  WRITELN(TEXT1,LINKAGE);  LINE_NO:=LINE_NO+1;
             GOTO 70;    END;
       IF (INDIC=2) AND (LNK=3) THEN
       BEGIN
           WRITELN(TEXT1,LINKAGE);
           WRITELN(TEXT1,TEMP_STMT);
           LINE_NO:=LINE_NO+2;
           GOTO 70;


       END;
       IF (INDIC=2) AND (LNK=2) THEN
       BEGIN
          IF TEMP_STMT[I]='B'
          THEN BEGIN
                  BUFER[I]:='B';  BUFER[I+1]:='Y';
                  TEMP_STMT[I]:=' ';  TEMP_STMT[I+1]:=' ';
               END
          ELSE BEGIN
                  BUFER[I]:='E'; BUFER[I+1]:='L';
                  BUFER[I+2]:='S'; BUFER[I+3]:='E';
                  BUFER[I+4]:=' '; BUFER[I+5]:='B';
                  BUFER[I+6]:='Y';
                  FOR J:=I TO I+5 DO TEMP_STMT[J]:=' ';
               END;
          WRITELN(TEXT1,BUFER);
          WRITELN(TEXT1,LINKAGE);
          WRITELN(TEXT1,TEMP_STMT);
          LINE_NO:=LINE_NO+3;
          GOTO 70;
       END;
       IF (INDIC=2) AND (LNK=1) THEN
       BEGIN
          FOR J:=1 TO ID_LENGTH DO  BUFER[J]:=RH_NAME[J];
          BUFER[9]:=':';
          WRITELN(TEXT1,LINKAGE);
          WRITELN(TEXT1,BUFER);
          WRITELN(TEXT1,TEMP_STMT);
          LINE_NO:=LINE_NO+3;
       END;
       70:;
    END;

 FUNCTION INDEX( VAR CH:CHAR ):INTEGER;
 (*  LOOK FOR THE FIRST SYMBOL=CH IN TEMP_STMT  *)
 (*  IF NOT FOUND RETURN ZERO, OTHERWISE RETURN *)
 (*  THE POSITION.                              *)
    VAR I:INTEGER;
    BEGIN
       INDEX:=0;
       FOR I:=TXT_PTR TO LINE_LENGTH DO
       BEGIN
          IF CH=TEMP_STMT[I] THEN
          BEGIN
             INDEX:=I;  GOTO 50;
          END;
       END;
       50:;
    END;

  FUNCTION REST_BLANK(VAR TX:INTEGER):BOOLEAN;
```

```
(*  REST_BLANK GETS TRUE VALUE IF THE REST OF   *)
(*  TEMP_STMT STARTING FROM TX IS BLANK.ELSE    *)
(*  RETURNS FALSE.                              *)
   VAR J:INTEGER;
   BEGIN
      REST_BLANK:=TRUE;
      FOR J:=TX TO LINE_LENGTH DO
      BEGIN
         IF TEMP_STMT[J] # ' ' THEN REST_BLANK:=FALSE;
      END;
   END;

PROCEDURE FIND PROC NAME;
(*  PREVIOUS TOKEN HAS BEEN 'PROCEDURE'.FIND   *)
(*  THE NEXT TOKEN WHICH IS THE PROCEDURE NAME*)
   VAR I,J:INTEGER;
   BEGIN
      WHILE (TEMP_STMT[TXT_PTR]=' ') DO TXT_PTR:=TXT_PTR+1;
      J:=1;
      REPEAT
         PROC_NAME[J]:=TEMP_STMT[TXT_PTR];
         J:=J+1;    TXT_PTR:=TXT_PTR+1;
      UNTIL NOT(TEMP_STMT[TXT_PTR] IN SYMBOLS)OR(J>ID_LENGTH);
      IF J<ID_LENGTH THEN
      BEGIN
         FOR I:=J TO ID_LENGTH DO PROC_NAME[I]:=' ';
      END;
   END;

PROCEDURE FIND ID;
(*  FIND A TOKEN.TRUNCATE IDENTIFIERS WITH   *)
(*  MORE THAN ID LENGTH=8. COMMENTS AND      *)
(*  STRING ASSIGNMENTS WILL BE RELEASED HERE*)
   VAR I,J:INTEGER;
   PROCEDURE  PASS_COMMENTS;
      VAR I:INTEGER;
    BEGIN  I:=TXT_PTR+1;
      IF NOT REST_BLANK(I) THEN
      BEGIN
            WHILE(TEMP_STMT[I]=' ') DO  I:=I+1;
            IF ((TEMP_STMT[I]='(') AND
               (TEMP_STMT[I+1]='*')) THEN
            BEGIN I:=I+1;
                  REPEAT  I:=I+1;
                  UNTIL    ((TEMP_STMT[I]='*') AND
                            (TEMP_STMT[I+1]=')'));
                  (* SKIP OVER COMMENT *)
                  TXT_PTR:=I+1;
            END;
      END;
    END;


   PROCEDURE PASS_QUOTES;
      VAR I,J:INTEGER;
      BEGIN I:=0;
         IF TEMP_STMT[TXT_PTR+1]='''' THEN I:=TXT_PTR+1
         ELSE IF TEMP_STMT[TXT_PTR+2]='''' THEN I:=TXT_PTR+2;
         IF I#0 THEN
         BEGIN
            I:=I+1;  J:=1;
            WHILE( J MOD 2 #0) DO
            BEGIN
               IF TEMP_STMT[I]='''' THEN
               BEGIN
                  WHILE( TEMP_STMT[I]='''') DO
                  BEGIN  J:=J+1; I:=I+1;  END;
               END
                  ELSE I:=I+1;
            END;
            (* SKIP STRING ASSIGNMENT *)
            TXT PTR:=I;
            PASS_QUOTES;
         END;
      END;

      BEGIN
      PASS_QUOTES;
      PASS_COMMENTS;
      IF REST_BLANK(TXT_PTR) THEN
      BEGIN
         SAVE_STMT;   READ_STMT;
      END;
      PREV TOKEN:=TOKEN;
      TOKEN:='                    ';
      I:=TXT PTR;
      (*  PASS DELIMETERS  *)
      WHILE((I<LINE_LENGTH)AND(NOT(TEMP_STMT[I] IN SYMBOLS)))
            DO   I:=I+1;
      IF I=LINE_LENGTH THEN
      BEGIN
         SAVE_STMT;   READ_STMT;
         I:=1;
         WHILE(NOT(TEMP_STMT[I] IN SYMBOLS)) DO I:=I+1;
      END;
      J:=1;
      WHILE((I<=LINE_LENGTH)AND(TEMP_STMT[I] IN SYMBOLS))DO
      BEGIN
         TOKEN[J]:=TEMP_STMT[I];
         I:=I+1;    J:=J+1;
      END;
      TXT_PTR:=I;
      FOR I:=1 TO ID_LENGTH DO ID[I]:=TOKEN[I];
      IF(TOKEN='TYPE                ') THEN DCL_TYPE:=FALSE;
```

```
        IF(TOKEN='ELSE_ERROR          ') THEN LAST_ELSE_ERROR:=
                                                  LINE_NO-1;
        IF((PREV_TOKEN='ELSE_ERROR           ') AND
           (TOKEN     ='END                 ') AND
           (TEMP_STMT[TXT_PTR]='.')) THEN SEMICOLON:=FALSE;
        IF(TOKEN='PROCEDURE           ') OR
          (TOKEN='FUNCTION            ') OR
          (TOKEN='BEGIN               ') OR
          (TOKEN='VAR                 ')THEN MORE_VAR:=FALSE;
       IF((TOKEN='READ                ') OR
          (TOKEN='READLN              ')) THEN
          READ_ASSIGNED:=TRUE;
       IF TEMP_STMT[TXT_PTR]='[' THEN
       BEGIN
         WHILE(TEMP_STMT[TXT_PTR] #']') DO TXT_PTR:=TXT_PTR+1;
         TXT_PTR:=TXT_PTR+1;
       END;
       IF TEMP_STMT[TXT_PTR]='.' THEN COT:=TRUE
                                 ELSE COT:=FALSE;
       IF TEMP_STMT[TXT_PTR]=';' THEN SEMI_COLON:=TRUE
                                 ELSE SEMI_COLON:=FALSE;
       IF (READ_ASSIGNED) AND (TEMP_STMT[TXT_PTR]=')') THEN
                  READ_ASSIGNED:=FALSE;
     END;

   PROCEDURE FIND_TYPE;
      VAR I,J,K,L:INTEGER;
      BEGIN
        IF (ID='END        ') AND (CODE=5) THEN GOTO 337;
        IF CODE=5 THEN CH:=':'  ELSE CH:='=';
        IF (INDEX(CH)=0) AND (MORE_VAR) THEN
        BEGIN
           SAVE_STMT;   READ_STMT;
           FIND_ID;
        END;
        IF MORE_VAR THEN
        BEGIN
           IF CODE=5 THEN CH:=':' ELSE CH:='='; I:=INDEX(CH)+1;
           WHILE(TEMP_STMT[I]=' ') DO I:=I+1;
           L:=1;
           WHILE(NOT(TEMP_STMT[I] IN [';',')'] )) DO
           BEGIN
              TIPE[L]:=TEMP_STMT[I];
              I:=I+1;  L:=L+1;
              IF REST_BLANK(I) THEN GOTO 85;
           END;
           85:;
           IF CODE=9 THEN TXT_PTR:=I;
           IF TIPE[1]='(' THEN BEGIN TIPE[L]:=')';L:=L+1;END;
           FOR K:=L TO TYPE_LENGTH DO TIPE[K]:=' ';
           FOR K:=1 TO ID_LENGTH DO TYP[K]:=TIPE[K];


           FOR I:=1 TO 3 DO
           BEGIN
            ST[I]:=TIPE[I]; FL[I]:=TIPE[I];
            AR[I]:=TIPE[I]; RC[I]:=TIPE[I];
           END;
           FL[4]:=TIPE[4];AR[4]:=FL[4];RC[4]:=FL[4];
           AR[5]:=TIPE[5];RC[5]:=AR[5];RC[6]:=TIPE[6];
        END;
         337:;
      END;

  (*  MAIN BODY OF GET_TOKEN  *)
  BEGIN
    CASE CODE OF
    1: BEGIN
        PROC_NAME:='OUT_MOST';  PRB:='PR';
        TOKEN:='PROCEDURE           ';
        FIRST_CALL:=FALSE;
        SAVE_STMT;   READ_STMT;
        END;
    2: SAVE_STMT;
    3: FIND_ID;
    4: FIND_PROC_NAME;
    5: BEGIN  FIND_ID;  FIND_TYPE;  END;
    6: IF TXT_PTR<LINE_LENGTH THEN
      IF TEMP_STMT[TXT_PTR]=')' THEN PROC_CALL:=FALSE;
    7: IF TXT_PTR<LINE_LENGTH THEN
      IF TEMP_STMT[TXT_PTR]='(' THEN PROC_CALL:=TRUE;
    9:  FIND_TYPE;
    8:
      BEGIN
       P:=TXT_PTR-1;
       WHILE((P>1) AND((TEMP_STMT[P] # ']') OR
                       (TEMP_STMT[P] # '['))) DO P:=P-1;
       IF READ_ASSIGNED THEN
          IF(P=1) OR (TEMP_STMT[P]=']')
               THEN ASSIGNED_VAR:=TRUE
               ELSE ASSIGNED_VAR:=FALSE;
       IF NOT READ_ASSIGNED THEN
       BEGIN
          CH:=':';
          IF INDEX(CH) # 0
               THEN ASSIGNED_VAR:=TRUE
               ELSE ASSIGNED_VAR:=FALSE;
       END
      END
     END
  END;

FUNCTION HASH (VAR ID_NAME:IDENTIFIER):INTEGER; FORWARD;
```

```
PROCEDURE  CREATE_PROC_NODE;
   VAR I,J:INTEGER;

   PROCEDURE FILL_SON_BROTHER;
       VAR J:INTEGER;
       (*  A PROCEDURE IS ADDED TO PROC/RB TREE   *)
       (*  MODIFY THE LINKS TO ITS BROTHER AND    *)
       (*  FATHER.                                *)
       BEGIN
           IF PROC_NAME # 'OUT_MOST' THEN
           BEGIN
              J:=PROC_RB_TREE[NEXT_PROC_NODE].FATHER;
              IF PROC_RB_TREE[J].FIRST_SON=0
              THEN PROC_RB_TREE[J].FIRST_SON:=NEXT_PROC_NODE
              ELSE BEGIN
                      J:=PROC_RB_TREE[J].FIRST_SON;
                      WHILE(PROC_RB_TREE[J].YOUNGER_BROTHER#0) DO
                          J:=PROC_RB_TREE[J].YOUNGER_BROTHER;
                      PROC_RB_TREE[J].YOUNGER_BROOTHER:=
                          NEXT_PROC_NODE;
                   END;
           END;
       END;

   (*  MAIN BODY OF CREATE PROC NODE  *)
   BEGIN
       WITH PROC_RB_TREE[NEXT_PROC_NODE] DO
       BEGIN
           (*  INSERT THE NODE INTO TREE  *)
           PR_RB_NAME:=PROC_NAME;
           INDICATOR:=PRB;
           PTR_TO_VDNODE:=NEXT_VD_NODE;
           NO_NLVAR_USED:=0;
           PTR_TO_NLVAR:=0;
           FIRST_SON:=0;
           YOUNGER_BROTHER:=0;
           MARK:='U';
           ENTRY:=LINE_NO+1;
           IF PRB='RB' THEN BEGIN LABLE:=GOTOLAB;
                                  GOTOLAB:=GOTOLAB+10;
                            END
                       ELSE LABLE:=0;
           IF PROC_NAME='OUT_MOST' THEN FATHER:=0
           ELSE BEGIN
                   (*  LINK THE NODE TO ITS FATHER  *)
                   FATHER_NAME:=STACK[PRED(TOP_OF_STACK)].NAME;
                   I:=HASH(FATHER_NAME);
                   J:=HASH_PTRS[I];
                   WHILE(FATHER_NAME # HASH_TABLE[J].PR_NAME) DO
                   BEGIN
                       REPEAT  I:=(I+1) MOD HASH_PTRS_LENGTH;


                       UNTIL   (HASH_PTRS[I] # 0);
                       J:=HASH_PTRS[I];
                   END;
                   FATHER:=HASH_TABLE[J].PTR_TO_PROC_TREE;
                END;
       END;
       FILL_SON_BROTHER;
       NEXT_PROC_NODE:=NEXT_PROC_NODE+1;
   END;

PROCEDURE POP_STACK;
(*  A PROCEDURE IS SCANNED,POP IT  *)
   BEGIN
       TOP_OF_STACK:=TOP_OF_STACK-1;
   END;

PROCEDURE PUSH_STACK;
   (*  A NEW PROCEDURE IS SEEN, PUSH IT  *)
   BEGIN
       STACK[TOP_OF_STACK].INDICATOR:=PRB;
       STACK[TOP_OF_STACK].NAME:=PROC_NAME;
       TOP_OF_STACK:=TOP_OF_STACK+1;
   END;

PROCEDURE INSERT_HASH_TABLE;
   VAR J:INTEGER;
   BEGIN
       J:=HASH(PROC_NAME);
       WHILE(HASH_PTRS[J] # 0) DO
          J:=(J+1) MOD HASH_PTRS_LENGTH;
       HASH_PTRS[J]:=HASH_INDEX;  J:=HASH_INDEX;
       HASH_INDEX:=HASH_INDEX+1;
       WITH HASH_TABLE[J] DO
       BEGIN
           PR_NAME:=PROC_NAME;
           PTR_TO_PROC_TREE:=PRED(NEXT_PROC_NODE);
           IF PROC_NAME='OUT_MOST' THEN SEG_ADR:=0
                                   ELSE SEG_ADR:=STMT_NO;
       END;
       LASTNODE:=PRED(NEXT_PROC_NODE);
   END;

FUNCTION  HASH;
   VAR  I,J,TEN,SUM: INTEGER;
   BEGIN
       (*  USING THE ORDINAL NUMBER OF SYMBOLS   *)
       (*  AND TREATING THE STRING AS A NUMBER   *)
       (*  HASH THE STRING.                      *)
       I:=1;   TEN:=1;   SUM:=0;
       WHILE(I<=ID_LENGTH) DO
       BEGIN
```

```
          J:=ORD(ID_NAME[I]);
          IF J>64 THEN J:=J-64 ELSE J:=J-32;
          SUM:=SUM+J*TEN;  TEN:=TEN*10;  I:=I+1;
      END;
      HASH:=SUM  MOD  HASH_PTRS_LENGTH;
   END;

PROCEDURE  VAR_DCL_PROCESSOR;
   PROCEDURE NEW_TYPE_NAME;
     VAR I:INTEGER;
     (*  IN CASE OF NESTED RECORDS,ARRAY,FILE  *)
     (*  SET,... A TYPE DECLARATION MUST BE    *)
     (*  GENERATED. THE TYPE NAME IS CONSIDERED*)
     (*  TO BE TP000001,TP000002,..............*)
     BEGIN
      NO_NEW_TYPE:=NO_NEW_TYPE+1;
      I:=ORD(TP_NAME[ID_LENGTH]);
      IF I=57 THEN
      BEGIN
       TP_NAME[ID_LENGTH]:='0';
       TP_NAME[ID_LENGTH-1]:=CHR(SUCC(ORD(TP_NAME[ID_LENGTH-1])));
      END
      ELSE TP_NAME[ID_LENGTH]:=CHR(SUCC(ORD(TP_NAME[ID_LENGTH])));
     END;

   PROCEDURE SAVE_SIMPLE(VAR ID:IDENTIFIER);
     BEGIN
      SIMPLE_TABLE[STT]:=ID;   STT:=STT+1;
     END;

   PROCEDURE SIMPLE_OR_RECORD(VAR II,JJ:INTEGER);
      (*  IN CASE OF ARRAY[...] OF...,IT MAY BE   *)
      (*  ARRAY[...]OF RECORD, OR ARRAY[...] OF   *)
      (*  SIMPLE TYPE,OR ARRAY[...]OF REC WHERE   *)
      (*  REC IS PREDECLARED AS STRUCTURE TYPE.   *)
      (*  JJ IS A CODE FOR RECOGNIZING THE CASE:  *)
      (*                                          *)
      (*  JJ=1 ---->  ARRAY[....]OF RECORD        *)
      (*  JJ=2 ---->  ARRAY[....]OF REC           *)
      (*  JJ=3 ---->  ARRAY[....]OF SIMPLE        *)
      (*  II WOULD BE A POINTER TO REC IN THE     *)
      (*  STRUCTURE TABLE.                        *)
      VAR I,J,K:INTEGER;
             T:IDENTIFIER;
      BEGIN
         I:=TYPE_LENGTH;
         WHILE(TIPE[I] IN [' ',';']) DO  I:=I-1;
         J:=I;
         WHILE  (TIPE[J] IN SYMBOLS)  DO
         BEGIN  J:=J-1;
           IF J=0 THEN BEGIN J:=1; GOTO 11; END;


         END;
         J:=J+1;
         11:;
         FOR K:=1 TO ID_LENGTH DO T[K]:=' ';  K:=1;
         WHILE((J<=I) AND (K<=ID_LENGTH)) DO
         BEGIN T[K]:=TIPE[J]; K:=K+1; J:=J+1; END;
         IF T='RECORD  ' THEN BEGIN JJ:=1; GOTO 12; END;
         K:=1;
         WHILE(K<=STRT-1) DO
         BEGIN
           IF STRUCTURE_TABLE[K].LEVEL<2 THEN
           BEGIN
              IF T=STRUCTURE_TABLE[K].NAME THEN
              BEGIN  JJ:=2;  II:=K;  GOTO 12;  END
              ELSE K:=K+1;
           END
           ELSE K:=K+1;
         END;
         JJ:=3;
         12:;
      END;

   PROCEDURE HANDLE_AR_ST_FL;
     VAR I,J:INTEGER;
     (*  IF TYPE IS SIMPLE ARRAY,FILE,SET,OR  *)
     (*  (...) THEN GENERATE A NEW TYPE NAME  *)
     (*  DECLARE A NEW TYPE AND USE THAT.     *)
     BEGIN
       FOR I:=1 TO LINE_LENGTH DO  BUFER[I]:=' ';
       NEW_TYPE_NAME;
       FOR I:=1 TO ID_LENGTH DO  BUFER[I]:=TP_NAME[I];
       BUFER[I]:='=';
       FOR J:=1 TO TYPE_LENGTH DO BUFER[I+J+1]:=TIPE[J];
       BUFER[I+J+1]:=';';
       WRITELN(TEXT2,BUFER);
       FOR J:=1 TO TYPE_LENGTH DO  TIPE[J]:=' ';
       FOR J:=1 TO ID_LENGTH DO  TYP[J]:=TP_NAME[J];
     END;
   PROCEDURE  GENERATE_DCL_STMT(VAR LC,HG:INTEGER);
      (*  WHEN A 'RECORD' IS COMPLETELY STORED   *)
      (*  IN STRUCTURE TABLE,CREATE A NEW TYPE   *)
      (*  NAME AND DECLARE IT AGAIN , THEN SAVE  *)
      (*  IT IN TEXT2.                           *)
      (*  EXAMPLE:                               *)
      (*          TP000001=RECORD                *)
      (*                        SAME COMPONENTS  *)
      (*                    END;                 *)
      (*                                         *)
      VAR I,J,K,L:INTEGER;
             LAST:BOOLEAN;
      BEGIN
```

```
       FOR I:=1 TO LINE_LENGTH DO  BUFER[I]:=' ';
       RC:='RECORD';  J:=LO;  GIVE_TYPE(TYP,J);
       FOR J:=1 TO ID_LENGTH DO  BUFER[J]:=TYP[J];
       BUFER[J]:='=';
       FOR I:=J+1 TO J+6 DO  BUFER[I]:=RC[I-J];
       WRITELN(TEXT2,BUFER);
       LO:=LO+1; I:=STRUCTURE_TABLE[LO].LEVEL;
       WHILE (LO <= HG) DO
       BEGIN
         FOR J:=1 TO LINE_LENGTH DO  BUFER[J]:=' ';
          (*  SKIP OVER THE INNER RECORDS  *)
          IF(STRUCTURE_TABLE[LO].LEVEL=I) THEN
          BEGIN
               ID:=STRUCTURE_TABLE[LO].NAME;  J:=LO;
               GIVE_TYPE(TYP,J);
               FOR J:=1 TO ID_LENGTH DO BUFER[J+15]:=ID[J];
               BUFER[J+15]:=':';
               FOR K:=1 TO ID_LENGTH DO BUFER[J+K+15]:=TYP[K];
               LAST:=TRUE;  L:=LO+1;
               (*  TAKE CARE OF SEMI COLON FOR THE  *)
               (*  LAST COMPONENT OF GENERATED      *)
               (*  RECORD TYPE.                     *)
               WHILE (L<=HG) DO
               BEGIN
                  IF STRUCTURE_TABLE[L].LEVEL=I
                  THEN LAST:=FALSE;
                 L:=L+1;
               END;
               IF (NOT LAST) THEN BUFER[J+K+15]:=';';
               WRITELN(TEXT2,BUFER);
          END;
          LO:=LO+1;
       END;
       FOR  K:=1 TO LINE_LENGTH DO  BUFER[K]:=' ';
       BUFER[10]:='E';  BUFER[11]:='N';  BUFER[12]:='D';
       BUFER[13]:=';';  WRITELN(TEXT2,BUFER);
  END;

PROCEDURE HANDLE_RECORD(VAR ID:IDENTIFIER);
   VAR II,JJ,I,J:INTEGER;
   BEGIN
      STK[TS]:=STRT;  TS:=TS+1;
      IF LEV=1 THEN
      BEGIN
         CASE  TYPSW  OF
         1: BEGIN
               (*  CALLED FROM TYPE PROCESSOR  *)
               WITH STRUCTURE_TABLE[STRT] DO
               BEGIN LEVEL:=LEV;  NAME:=ID;
                     IND1:=' ';  IND2:='S';
                     TCODE:=STT;


              END;
              NEW_TYPE_NAME;  SAVE_SIMPLE(TP_NAME);
           END;
        2: BEGIN
              (*  CALLED FROM VAR_DCL_PROCESSOR  *)
              (*                                 *)
              (*  IF IT HAS BEEN ARRAY[...] OF   *)
              (*  RECORD .THEN REPLACE 'RRECORD' *)
              (*  WITH A NEW TYPE NAME AND USE   *)
              (*  THIS NAME FOR CREATING THE NEW *)
              (*  DECLARATION.                   *)
              (*  EXAMPLE:                       *)
              (*        ARRAY[...]OF RECORD      *)
              (*     IS CHANGED TO:              *)
              (*        ARRAY[...]OF TP000002    *)
              (*        TP000002=RECORD          *)
              (*                  ....           *)
              (*                END;             *)
              (*                                 *)
              IF ASFR=3 THEN
              BEGIN I:=TYPE_LENGTH;
                WHILE(TIPE[I] IN [' ',';']) DO I:=I-1;
                J:=I;
                WHILE(TIPE[J] IN SYMBOLS) DO
                BEGIN TIPE[J]:=' '; J:=J-1; END;
                NEW_TYPE_NAME;
                FOR I:=1 TO ID_LENGTH DO TIPE[I+J]:=TP_NAME[I];
                WITH STRUCTURE_TABLE[STRT] DO
                BEGIN  LEVEL:=LEV; NAME:=TP_NAME;
                       IND1:=' ';  IND2:=' T; TCODE:=0;
                END;
                TI:=TIPE; GOTO 348;
              END;
               NEW_TYPE_NAME;
               WITH STRUCTURE_TABLE[STRT] DO
               BEGIN LEVEL:=LEV;  NAME:=TP_NAME;
                     IND1:=' ';  IND2:=' 'T
                     TCODE:=0;
               END
           END
        END
     END ELSE
     BEGIN
        NEW_TYPE_NAME;
        WITH STRUCTURE_TABLE[STRT] DO
        BEGIN  LEVEL:=LEV;  NAME:=ID;  IND1:=' ';
               IND2:='S';  TCODE:=STT;
        END;
        SAVE_SIMPLE(TP_NAME);
     END;
     348::
        STRT:=STRT+1;  LEV:=LEV+1;
        CODE:=5;  GET TOKEN;
        WHILE( ID#'END      ') DO
        BEGIN
           IF ((RC='PACKED') OR (ST='SET') OR
              (FL='FILE') OR (TYP[1]='(')) THEN
           BEGIN
              WITH STRUCTURE_TABLE[STRT] DO
              BEGIN  LEVEL:=LEV; NAME:=ID; IND1:=' ';
                     IND2:='S';  TCODE:=STT;
              END;
              STRT:=STRT+1;  HANDLE_AR_ST_FL;
              SAVE_SIMPLE(TYP);  GOTO 347;
           END;
           IF AR='ARRAY' THEN
           BEGIN
              SIMPLE_OR_RECORD(II,JJ);
              CASE  JJ  OF
              1: BEGIN
                 (*  RESERVED  *)
                 END;
              2: BEGIN
                     (*  ARRAY[...]OF REC, REC PREDCL  *)
                     WITH STRUCTURE_TABLE[STRT] DO
                     BEGIN LEVEL:=LEV; NAME:=ID;
                           IND1:=' ';  IND2:='R';
                           TCODE:=II;
                     END;
                 END;
              3: BEGIN
                     (*  ARRAY[...] OF SIMPLE TYPE  *)
                     WITH STRUCTURE_TABLE[STRT] DO
                     BEGIN LEVEL:=LEV; NAME:=ID;
                           IND1:=' ';  IND2:='S';
                           TCODE:=STT;
                     END;
                     HANDLE_AR_ST_FL;  SAVE_SIMPLE(TYP);
                 END
              END;
              STRT:=STRT+1;  GOTO 347;
           END;
           IF RC='RECORD' THEN
           BEGIN  HANDLE_RECORD(ID);  GOTO 347;  END;
           SIMPLE_OR_RECORD(II,JJ);
           IF JJ=2 THEN
           BEGIN
              WITH STRUCTURE_TABLE[STRT] DO
              BEGIN  LEVEL:=LEV; NAME:=ID;  IND1:=' ';
                     IND2:='R';  TCODE:=II;
              END;
              GOTO 347;


          END;
          WITH STRUCTURE_TABLE[STRT] DO
          BEGIN  LEVEL:=LEV;  NAME:=ID;  IND1:=' ';
                 IND2:='S';   TCODE:=STT;
          END;
          SAVE_SIMPLE(TYP);   STRT:=STRT+1;
          347:  GET_TOKEN;
       END;
       LEV:=LEV-1;
       I:=STRT-1;  J:=STK[TS-1];  TS:=TS-1;
       GENERATE_DCL_STMT(J,I);
   END;

PROCEDURE  CREATE_VD_NODE;
   VAR I,J:INTEGER;

   PROCEDURE  FILLUP_SON_BROTHER;
       VAR K:INTEGER;
       BEGIN
          IF PROC_NAME # 'OUT_MOST' THEN
          BEGIN
              K:=VAR_DCL_TREE[NEXT_VD_NODE].FATHER;
              IF VAR_DCL_TREE[K].FIRST_SON=0
              THEN VAR_DCL_TREE[K].FIRST_SON:=NEXT_VD_NODE
              ELSE BEGIN
                      K:=VAR_DCL_TREE[K].FIRST_SON;
                      WHILE(VAR_DCL_TREE[K].YOUNGER_BROTHER#0)DO
                            K:=VAR_DCL_TREE[K].YOUNGER_BROTHER;
                      VAR_DCL_TREE[K].YOUNGER_BROTHER:=
                                             NEXT_VD_NODE;
                   END;
          END;
       END;

   BEGIN
        WITH VAR_DCL_TREE[NEXT_VD_NODE] DO
        BEGIN
           PTR_TO_VDTABLE:=NEXT_VD_ROW;
           SIZE:=0;
           PR_NAME:=PROC_NAME;
           FIRST_SON:=0;
           YOUNGER_BROTHER:=0;
           IF PROC_NAME='OUT_MOST' THEN FATHER:=0
           ELSE BEGIN
                    FATHER_NAME:=STACK[TOP_OF_STACK-2].NAME;
                    I:=HASH(FATHER_NAME);
                    J:=HASH_PTRS[I];
                    WHILE(FATHER_NAME # HASH_TABLE[J].PR_NAME)DO
                    BEGIN
                          REPEAT  I:=(I+1) MOD HASH_PTRS_LENGTH;
                          UNTIL   (HASH_PTRS[I] # 0);
                          J:=HASH_PTRS[I];
                    END;
                    J:=HASH_TABLE[J].PTR_TO_PROC_TREE;
                    FATHER:=PROC_DB_TREE[J].PTR_TO_VDNODE;
                END;
      END;
      FILLUP_SON_BROTHER;
      NEXT_VD_NODE:=NEXT_VD_NODE+1;
  END;

PROCEDURE  FILLUP_VD_TABLE;
    VAR I,II,JJ:INTEGER;
    PROCEDURE  INSERT_INTO_VDTAB;
       BEGIN
          CASE  ASFR  OF
          1: BEGIN
                 (*  SIMPLE VAR  *)
                 WITH  VAR_DCL_TABLE[NEXT_VD_ROW] DO
                 BEGIN  VNAME:=ID; INDIC:=ASFR; TCODE:=STT;
                 END;
                 SAVE_SIMPLE(TYP);
              END;
          2,3: BEGIN
                 (*  SIMPLE STRUCTURE VAR, NOT  *)
                 (*  ARRAY OF RECORDS.          *)
                 WITH VAR_DCL_TABLE[NEXT_VD_ROW] DO
                 BEGIN VNAME:=ID; INDIC:=ASFR; TCODE:=STRT;
                 END;
              END;
          5: BEGIN
              (*  SIMPLE VAR,STRUCTURE TYPE. PREDCL.  *)
              WITH VAR_DCL_TABLE[NEXT_VD_ROW] DO
              BEGIN  VNAME:=ID; INDIC:=ASFR; TCODE:=II;
              END;
             END;
          4: BEGIN
                 (*  ARRAY[...]OF REC,REC PREDCL.  *)
                 WITH  VAR_DCL_TABLE[NEXT_VD_ROW] DO
                 BEGIN  VNAME:=ID; INDIC:=ASFR; TCODE:=II;
                         TTCODE:=STT;
                         HANDLE_AR_ST_FL;
                         SAVE_STMPCE(TP_NAME);
                 END;
              END
          END;
          NEXT_VD_ROW:=NEXT_VD_ROW+1;
          VAR_DCL_TREE[NEXT_VD_NODE-1].SIZE:=
          VAR_DCL_TREE[NEXT_VD_NODE-1].SIZE+1;
       END;
       (*  MAIN BODY OF FILLUP_VD_TAB  *)
       BEGIN


          CODE:=3;   GET_TOKEN;
          REPEAT
             IF (ST='SET') OR (FL='FILE') OR
                (RC='PACKED') OR (TYP[1]='(')
             THEN BEGIN
                       HANDLE_AR_ST_FL; ASFR:=1;
                       INSERT_INTO_VCTAB;
                   END
             ELSE IF AR='ARRAY' THEN
             BEGIN
                SIMPLE_OR_RECORD(II,JJ);
                CASE  JJ  OF
                1: BEGIN
                        ASFR:=3; INSERT_INTO_VDTAB;
                        LEV:=1;  TS:=1;
                        HANDLE_RECORD(ID);
                        VAR_DCL_TABLE[NEXT_VD_ROW-1].TTCODE:=STT;
                        TIPE:=TT;
                        HANDLE_AR_ST_FL;
                        SAVE_STMPCE(TP_NAME);
                    END;
                2: BEGIN
                        ASFR:=4;  INSERT_INTO_VDTAB;
                    END;
                3: BEGIN
                        ASFR:=1;  HANDLE_AR_ST_FL;
                        INSERT_INTO_VDTAB;
                    END
                END;
             END
             ELSE IF RC='RECORD' THEN
             BEGIN
                ASFR:=2;  INSERT_INTO_VDTAB;
                LEV:=1;  TS:=1;
                HANDLE_RECORD(ID);
             END
             ELSE BEGIN
                       SIMPLE_OR_RECORD(II,JJ);
                       IF JJ=2 THEN
                       BEGIN  ASFR:=5; INSERT_INTO_VDTAB;
                       END ELSE
                       BEGIN  ASFR:=1; INSERT_INTO_VDTAB;
                       END;
                   END;
             GET_TOKEN;
          UNTIL NOT(MORE_VAR);
      END;

   PROCEDURE  TYPE_PROCESSOR;
      VAR II,JJ:INTEGER;
      BEGIN
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
               TYPSW:=1; CODE:=3;  GET_TOKEN;
               REPEAT
                  CODE:=9;  GET_TOKEN;
                  IF ((RC='PACKED') OR (ST='SET') OR
                     (FL='FILE') OR (TYP[1]='('))
                  THEN SAVE_SIMPLE(ID)
                  ELSE IF AR='ARRAY' THEN
                  BEGIN
                     SIMPLE_OR_RECORD(II,JJ);
                     CASE  JJ  OF
                     1: BEGIN
                           WITH STRUCTURE_TABLE[STRT] DO
                           BEGIN LEVEL:=0; NAME:=ID; IND1:='A';
                                 IND2:='R'; TCODE:=STRT+1;
                           END;
                           STRT:=STRT+1;  NEW_TYPE_NAME;
                           LEV:=1;  TS:=1;
                           HANDLE_RECORD(TP_NAME);
                        END;
                     2: BEGIN
                           WITH STRUCTURE_TABLE[STRT] DO
                           BEGIN LEVEL:=0; NAME:=ID;
                                 IND1:='A'; IND2:='R';
                                 TCODE:=II;
                           END;
                           STRT:=STRT+1;
                        END;
                     3: SAVE_SIMPLE(ID)
                     END;
                  END
                  ELSE IF RC='RECORD' THEN
                  BEGIN LEV:=1;  TS:=1;  HANDLE_RECORD(ID); END
                                        ELSE SAVE_SIMPLE(ID);
                  CODE:=3;  GET_TOKEN;
               UNTIL(NOT MORE_VAR);
               TYPSW:=2;
           END;

   (*  MAIN BODY OF VAR_DCL_PROCESSOR  *)
   BEGIN
       CREATE_VD_NODE;
       VAR_DCL_TABLE[NEXT_VD_ROW].VNAME:=PROC_NAME;
       VAR_DCL_TREE[NEXT_VD_NODE-1].SIZE:=
       VAR_DCL_TREE[NEXT_VD_NODE-1].SIZE+1;
       NEXT_VD_ROW:=NEXT_VD_ROW+1;
       MORE_VAR:=TRUE;
       CODE:=3;  GET_TOKEN;
       WHILE ((MORE_VAR) AND (ID #'TYPE      '))DO GET_TOKEN;
       IF ID='TYPE      ' THEN  TYPE_PROCESSOR;
       WHILE (TOKEN='VAR                   ') DO


       BEGIN
          IF POS_NEW_TYPE=-1 THEN POS_NEW_TYPE:=STMT_NO-1;
          MORE_VAR:=TRUE;
          FILLUP_VD_TABLE;
       END;
   END;

PROCEDURE NL_VAR_PROCESSOR(VAR NV:IDENTIFIER;VAR TV:STRING2);
FORWARD;
PROCEDURE LOOKUP_PROC_NAMES(VAR PROC_NAME_FOUND:BOOLEAN);
   VAR I,J,K:INTEGER;
   (*  CHECK IF IDENTIFIER IS A PROCEDURE  *)
   (*  NAME. IF SO,SAVE IT IN NL_VAR_TABLE *)
   BEGIN
      PROC_NAME_FOUND:=FALSE;
      J:=HASH(ID);    I:=J;
      K:=HASH_PTRS[J];
      IF K=0 THEN GOTO 500;
      WHILE(ID # HASH_TABLE[K].PR_NAME) DO
      BEGIN
         REPEAT  J:=(J+1) MOD HASH_PTRS_LENGTH;
         UNTIL   (HASH_PTRS[J] # 0);
         K:=HASH_PTRS[J];
         IF I=J THEN GOTO 500;
      END;
      PVDT:=HASH_TABLE[K].PTR_TO_PROC_TREE;
      IDT:='PR';   NL_VAR_PROCESSOR(ID,IDT);
      CODE:=7;   GET_TOKEN;  PROC_NAME_FOUND:=TRUE;
      500:;
   END;

PROCEDURE CURRENT_PROC(VAR L:INTEGER);
   VAR I,J:INTEGER;
   (*  GIVE A POINTER TO PROC/RB TREE TO THE   *)
   (*  CURRENT PROCEDURE WHERE IDENTIFIER IS   *)
   (*  FOUND IN ITS BODY.                      *)
   BEGIN
      PROC_NAME:=STACK[TOP_OF_STACK-1].NAME;
      J:=HASH(PROC_NAME);
      I:=HASH_PTRS[J];
      WHILE(PROC_NAME#HASH_TABLE[I].PR_NAME) DO
      BEGIN
         REPEAT  J:=(J+1) MOD HASH_PTRS_LENGTH;
         UNTIL   (HASH_PTRS[J] #0);
         I:=HASH_PTRS[J];
      END;
      L:=HASH_TABLE[I].PTR_TO_PROC_TREE;
   END;

PROCEDURE DCL_AREA (VAR L,LO,HI:INTEGER);
   VAR I,J:INTEGER;
```

```
    (*  GIVE LOWER AND HIGHER POINTERS TO  *)
    (*  DECLARATION AREA OF PROCEDURE      *)
    (*  POINTED TO BY L .                  *)
    BEGIN
       J:=PROC_RB_TREE[L].PTR_TO_VDNODE;
       LO:=VAR_DCL_TREE[J].PTR_TO_VDTABLE;
       HI:=VAR_DCL_TREE[J].SIZE+LO;
    END;

PROCEDURE SEARCH_VD_TBL( VAR L,PTVDTBL:INTEGER);
    VAR  I,J,K:INTEGER;
    (*  SEARCH VAR_DCL_TABLE FOR IDENTIFIER  *)
    (*  TO SEE IF ID IS GLOBAL OR LOCAL.     *)
    BEGIN K:=L;
       REPEAT
          DCL_AREA(L,J,I);
          WHILE (J<I) DO
          BEGIN
             IF ID=VAR_DCL_TABLE[J].VNAME THEN GOTO 697;
             J:=J+1;
          END;
          L:=PROC_RB_TREE[L].FATHER;
       UNTIL  (L=0);
       (*  IDENTIFIER IS NOT FOUND AT ALL.  *)
       PTVDTBL:=0;
       GOTO 698;
       697:;
       (*  IF PTVDTAB<0  ----> IDENTIFIER IS LOCAL  *)
       (*  IF PTVDTAB>0  ----> IDENTIFIER IS GLOBAL *)
       (*  IF PTVDTAB=0  ----> IDENTIFIER NOT FOUND *)
       IF K=L THEN PTVDTBL:=-J  ELSE PTVDTBL:=J;
       698:;
    END;
PROCEDURE  SEARCH_COMPONENTS (VAR J:INTEGER; VAR FOUND:BOOLEAN);
    VAR M:INTEGER;
    (*  SEARCH STRUCTURE TABLE TO SEE IF  *)
    (*  IDENTIFIER IS A COMPONENT OS SOME *)
    (*  STRUCTURE VARIABLE.               *)
    BEGIN
       IF J<0 THEN BEGIN J:=-J; J:=J+1; END
              ELSE J:=VAR_DCL_TABLE[J].TCODE+1;
       FOUND:=FALSE; M:=STRUCTURE_TABLE[J].LEVEL;
       WHILE (M<=STRUCTURE_TABLE[J].LEVEL) DO
       BEGIN
          IF STRUCTURE_TABLE[J].LEVEL=M THEN
             IF STRUCTURE_TABLE[J].NAME=ID THEN
             BEGIN  FOUND:=TRUE;  GOTO 854;  END;
             J:=J+1;
       END;
       854:;
    END;


PROCEDURE STRUC_VAR_HANDLING;
    VAR L,PTVDTBL,I,J:INTEGER;
               FOUND:BOOLEAN;
    BEGIN
       CURRENT_PROC(L);
       SEARCH_VD_TBL(L,PTVDTBL);
       (*  A STRUCTURE VARIABLE IS GLOBAL,SAVE IT  *)
       (*  IN THE WITH STACK.                      *)
       IF PTVDTBL>0 THEN
       BEGIN  WITH_STACK[TWS,1]:=PTVDTBL;
              WITH_STACK[TWS,2]:=0;
              TWS:=TWS+1;
       END;
       (*  IDENTIFIER IS NOT FOUND IN VAR_DCL_TABLE  *)
       (*  SO IT MIGHT BE A COMPONENT OF SOME        *)
       (*  STRUCTURE VARIABLE. THEREFORE,SEARCH THE  *)
       (*  STRUCTURE TABLE,IF IT WAS FOUND THERE,    *)
       (*  SAVE IT IN WITH STACK. THE SECOND COLUMN  *)
       (*  OF WITH STACK IS IN A SENSE A KIND OF     *)
       (*  DYNAMIC LINK............................. *)
       IF PTVDTBL=0 THEN
       BEGIN
          I:=TWS-1;
          WHILE (I>=1) DO
          BEGIN
             J:=WITH_STACK[I,1];
             SEARCH_COMPONENTS(J,FOUND);
             IF FOUND THEN
             BEGIN
                WITH_STACK[TWS,1]:=-J;
                WITH_STACK[TWS,2]:=I;
                TWS:=TWS+1;
                GOTO 131;
             END;
             I:=I-1;
          END;
          131:;
       END;
    END;

FUNCTION CHANGABLE : BOOLEAN;
    (*  A VARIABLE IS CHANGABLE IF:     *)
    (*                                  *)
    (*  1. SOMETHING IS ASSIGNED TO IT  *)
    (*  2. APPEARS IN A READ STATEMENT  *)
    (*  3. APPEARS IN A CALL TO A       *)
    (*     PROCEDURE AS AN ARGUMENT.    *)
    BEGIN
       CODE:=8;  GET_TOKEN;
       CHANGABLE:=(ASSIGNED_VAR) OR (PROC_CALL);
    END;

PROCEDURE GLOBAL(VAR  NONLOCAL:BOOLEAN;VAR JJ,II:INTEGER);
   (*  CHECKS WHETHER A VARIABLE IS GLOBAL OR NOT.   *)
   (*  FIRST BY LOOKING AT COMPONENTS,THEN            *)
   (*  VAR DCL TABLE. THAT IS TO SAY, IF WITH         *)
   (*  STATEMENT IS USED, THE PRIORITY IS WITH THE   *)
   (*  COMPONENT................................... *)
    VAR I,J,L,P:INTEGER;
        FOUND:BOOLEAN;
    BEGIN
       NONLOCAL:=FALSE;
       IF TWS>1 THEN
       BEGIN
          I:=TWS-1;
          WHILE( I>=1 )DO
          BEGIN
             J:=WITH STACK[I,1];
             SEARCH COMPONENT(J,FOUND);
             IF FOUND THEN
             BEGIN
                NONLOCAL:=TRUE; JJ:=-J; II:=I; GOTO 132;
             END;
             I:=I-1;
          END;
          132:;
       END
       ELSE BEGIN
              (* NOT INSIDE OF WITH STATEMENT *)
              CURRENT PROC(L);
              SEARCH VO TBL(L,P);
              IF  P>0  THEN
              BEGIN
                 NONLOCAL:=TRUE;  JJ:=P;
              END;
           END;
    END;

PROCEDURE CHNG_NL_VAR;
    VAR II,JJ,I:INTEGER;
       NONLOCAL:BOOLEAN;
    BEGIN
       (*  SAVE ANY CHANGABLE NONLOCAL VARIABLE  *)
       (*  IN  NL_VAR TABLE BY CALLING THE        *)
       (*  NL_VAR_PROCESSOR...................... *)
       IF CHANGABLE THEN
       BEGIN
          GLOBAL(NONLOCAL,JJ,II);
          IF NONLOCAL THEN
          BEGIN
             IF JJ>0 THEN


            BEGIN
               (* NONLOCAL SIMPLE VARIABLE *)
               PVDT:=JJ;  IDT:='ID';  NL_VAR_PROCESSOR(ID,IDT);
            END;
            IF JJ<0 THEN
            BEGIN
               (* NONLOCAL STRUCTURE VARIABLE *)
               PVDT:=-JJ; IDT:='CD';  NL_VAR_PROCESSOR(ID,IDT);
               I:=II;
               WHILE (WITH_STACK[I,2]#0) DO
               BEGIN
                  PVDT:=-WITH_STACK[I,1]; IDT:='CD';
                  ID:='        ';  NL_VAR_PROCESSOR(ID,IDT);
                  I:=WITH_STACK[I,2];
               END;
               PVDT:=WITH STACK[I,1];  IDT:='ID';
               NL_VAR_PROCESSOR(ID,IDT);
            END;
         END;
      END;
    END;

PROCEDURE NL VAR PROCESSOR;
    (*  SAVE THE NONLOCAL CHANGABLE VARIABLE.     *)
    (*                                            *)
    (*  IF INDICATOR='ID' ---->SIMPLE VAR         *)
    (*  IF INDICATOR='RB' ---->RECOVERY BLOCK     *)
    (*  IF INDICATOR='PR' ---->PROCEDURE          *)
    (*  IF INDICATOR='CD' ---->COMPONENT          *)
    (*  EXAMPLE:                                  *)
    (*          P.X.Y  IS STORED AS:              *)
    (*          -----                             *)
    (*     'CD',---->TO (Y) IN STRUCTURE TABLE   *)
    (*     'CD',---->TO (X) IN STRUCTURE TABLE   *)
    (*     'ID',---->TO (P) IN VAR_DCL_TABLE     *)
    (*                                            *)
    (*  REPEATITION IS TAKEN CARE OF........... *)
    VAR I,J,M:INTEGER;
       FLAG:BOOLEAN;
    BEGIN
       PROC_NAME:=STACK[TOP_OF_STACK-1].NAME;
       IF PROC_NAME='OUT MOST' THEN GOTO 551;
       M:=HASH(PROC_NAME);
       I:=HASH_PTRS[M];
       WHILE (PROC_NAME # HASH_TABLE[I].PR_NAME) DO
       BEGIN
          REPEAT  M:=(M+1) MOD HASH_PTRS_LENGTH;
          UNTIL   (HASH_PTRS[M] # 0);
          I:=HASH_PTRS[M];
       END;
       M:=I;
```

```
        M:=HASH_TABLE[M].PTR_TO_PROC_TREE;
         IF PROC_RB_TREE[M].PTR_TO_NLVAR=0 THEN
           PROC_RB_TREE[M].PTR_TO_NLVAR:=NEXT_NL_VAR;
           IF TV='CD' THEN GOTO 549;
           J:=PROC_RB_TREE[M].NO_NLVAR_USED;
         I:=PROC_RB_TREE[M].PTR_TO_NLVAR;   J:=J+I;
         WHILE(I<J) DO
         BEGIN
            IF I=1 THEN FLAG:=TRUE
            ELSE FLAG:=(NL_VAR_TABLE[I-1].INDICATOR#'CD');
            IF (NL_VAR_TABLE[I].PTR_TO_VOTAB=PVOT) AND
               (NL_VAR_TABLE[I].INDICATOR=TV) AND
                 FLAG  THEN GOTO 551
                       ELSE I:=I+1;
         END;
         549:;
         NL_VAR_TABLE[NEXT_NL_VAR].INDICATOR:=TV;
         NL_VAR_TABLE[NEXT_NL_VAR].PTR_TO_VOTAB:=PVOT;
         PROC_RB_TREE[M].NO_NLVAR_USED:=
         PROC_RB_TREE[M].NO_NLVAR_USED+1;
         NEXT_NL_VAR:=NEXT_NL_VAR+1;
         551:;
     END;
PROCEDURE  BACKTRACK_NLVAR;
    (*  NONLOCAL VARIABLE AREA OF NESTED   *)
    (*  PROCEDURES IN NL_VAR_TABLE ARE     *)
    (*  NESTED AS WELL.                    *)
    (*  BY CALLING THIS PROCEDURE AND      *)
    (*  USING A BACKTRACKING STRATEGY      *)
    (*  NL_VAR_TABLE WILL BE REARRANGED    *)
    (*  AND THE POINTERS WILL BE MODIFIED  *)
    (*  IN PROC/RB_TREE.................   *)
    VAR I,LO,HI,M:INTEGER;
    BEGIN
       CURRENT_PROC(M);
       LO:=PROC_RB_TREE[M].PTR_TO_NLVAR;
       HI:=PROC_RB_TREE[M].NO_NLVAR_USED+LO-1;
       IF  LO#0  THEN
       BEGIN
          PROC_RB_TREE[M].PTR_TO_NLVAR:=NEXT_NL;
          FOR I:=LO TO HI DO
          BEGIN
            NL_BUFER[NEXT_NL].INDICATOR:=
            NL_VAR_TABLE[I].INDICATOR;
            NL_BUFER[NEXT_NL].PTR_TO_VOTAB:=
            NL_VAR_TABLE[I].PTR_TO_VOTAB;
            NEXT_NL:=NEXT_NL+1;
          END;
          NEXT_NL_VAR:=LO;
       END;
    END;



PROCEDURE WITH_BLOCK_PROCESSOR; FORWARD;
PROCEDURE REC_BLOCK_PROCESSOR; FORWARD;
(***********************************************)
(*                BODY PROCESSOR               *)
(*                                             *)
(*  BODY PROCESSOR GOES THRU THE BODY OF THE   *)
(*  HITTED PROCEDURE. IT WILL LOOK FOR THOSE   *)
(*  VARIABLES NONLOCAL TO THE PROCEDURE AND    *)
(*  CHANGABLE AS WELL. IF THERE ARE SUCH VARS  *)
(*  THEY WILL BE SAVED IN NL_VAR_TABLE.        *)
(*  IF THERE ARE SOME PROCEDURE CALLS IN THE   *)
(*  PROCEDURE BODY,THEY WILL BE SAVED IN THE   *)
(*  SAME TABLE TOO. INOTHER WORDS, ANY CALLED  *)
(*  PROCEDURE MAY CHANGE SOME OF THE NONLOCAL  *)
(*  VARIABLES AS WELL.                         *)
(*  THIS PROCEDURE IS RECURSIVELY CALLED IN    *)
(*  CASE OF NESTED BLOCKS.                     *)
(*  BY SEEING RECOVERY BLOCKS OR WITH          *)
(*  STAEMENTS ,CORRESPONDING PROCEDURES WILL   *)
(*  BE CALLED. THESE PROCEDURES MAY CALL THE   *)
(*  BODY PROCESSOR IF A NEW BLOCK HAS STARTED. *)
(*  CASE STAEMENTS ARE TREATED AS BLOCKS.......*)
(*                                             *)
(***********************************************)
PROCEDURE BODY_PROCESSOR;
    VAR PROC_NAME_FOUND:BOOLEAN;
        KW:INTEGER;
    BEGIN
       CODE:=3;    GET_TOKEN;
       WHILE (TOKEN #'END                    ') DO
       BEGIN
            IF TOKEN='BEGIN                  ' THEN
               BEGIN
                  BODY_PROCESSOR;  GOTO 301;
               END;
            IF TOKEN='ENSURE                 ' THEN
               BEGIN
                  INDIC:=2;  LNK:=1;
                  REC_BLOCK_PROCESSOR; PCP_STACK; GOTO 301;
               END;
               IF TOKEN='WITH                   ' THEN
               BEGIN WITH_BLOCK_PROCESSOR; GOTO 301; END;
            IF TOKEN='CASE                   ' THEN
            BEGIN  BODY_PROCESSOR; GOTO 301; END;
            IF DOT THEN
            BEGIN
               (*  WHEN DOT IS THE VALUE TRUE, THEN  *)
               (*  STRUCTURE VARIABLE HAS THE FORM:  *)
               (*    XXX.XXXX.XX.....................*)
```

```
             WC:=0;
             WHILE  DOT  DO
             BEGIN  DOT:=FALSE;
                    STRUC_VAR_HANDLING;
                    WC:=WC+1;  CODE:=3;   GET_TOKEN;
             END;
             CHNG_NL_VAR;
             TWS:=TWS-WC;   GOTO 300;
          END;
          CHNG_NL_VAR;
          IF NOT( CHANGABLE ) THEN
             LOOKUP_PROC_NAMES(PROC_NAME_FOUND);
          300::
                 CODE:=3;    GET_TOKEN;
           301::
      END;
      CODE:=3;  GET_TOKEN;
   END;

(***************************************)
(*          REC_BLOCK_PROCESSOR        *)
(*                                     *)
(*  RECOVERY BLOCK IS CALLED WHENEVER  *)
(*  AN 'ENSURE' STAEMENT HAS BEEN SEEN *)
(*  IN THE BODY OF A PROCEDURE.        *)
(*  ANY RECOVERY BLOCK WILL BE TREATED *)
(*  AS IF IT WERE A PROCEDURE. A NAME  *)
(*  WILL BE CREATED AND ASSIGNED TO    *)
(*  THAT,A NODE IN PROC/RB TREE WILL   *)
(*  BE INSERTED WITH ALL INFORMATION   *)
(*  ABOUT THE RECOVERY BLOCK.          *)
(*  IN THE CASE OF NESTED RECOVERY     *)
(*  BLOCKS, THE PROCESSOR IS CALLED    *)
(*  RECURSIVELY. THE REST OF IT IS     *)
(*  SIMILAR TO BODY PROCESSOR......... *)
(*                                     *)
(***************************************)
PROCEDURE REC_BLOCK_PROCESSOR;
   VAR PROC_NAME_FOUND:BOOLEAN;
       I,J:INTEGER;


   PROCEDURE CREATE_RB_NAME;
    (*  NEW NAMES FOR RECOVERY BLOCKS  *)
    (*  ARE CONSIDERED TO BE LIKE:     *)
    (*  RB000001,RB000002,RB000003,....*).
    (*  THEY WILL BE CREATED WHENEVER  *)
    (*  A RECOVERY BLOCK IS SEEN.......*)
    VAR I:INTEGER;
    BEGIN
     I:=ORD(RB_NAME[ID_LENGTH]);


     IF I=57 THEN
     BEGIN
      RB_NAME[ID_LENGTH]:='0';
      RB_NAME[ID_LENGTH-1]:=CHR(SUCC(ORD(RB_NAME[ID_LENGTH-1])));
     END
     ELSE RB_NAME[ID_LENGTH]:=CHR(SUCC(ORD(RB_NAME[ID_LENGTH])));
    END;

   (*    MAIN BODY OF REC_BLOCK_PROCESSOR    *)
   BEGIN
          CREATE_RB_NAME;    IDT:='RB';
          PVDT:=NEXT_PROC_NODE;
          NL_VAR_PROCESSOR(RB_NAME,IDT);
          PROC_NAME:=RB_NAME;        PRB:='RB';
          CREATE_PROC_NODE;
          PUSH_STACK;
          INSERT_HASH_TABLE;
          CODE:=3;      GET_TOKEN;
          WHILE (TOKEN # 'ELSE_ERROR           ') DO
          BEGIN
              IF TOKEN='BEGIN                  ' THEN
                BEGIN
                  BODY_PROCESSOR;    GOTO 801;
                END;
              IF TOKEN='ENSURE                 ' THEN
                BEGIN
                  INDIC:=2;  LNK:=1;
                  REC_BLOCK_PROCESSOR; POP_STACK; GOTO 800;
                END;
              IF(TOKEN='BY                     ') OR
                (TOKEN='ELSE_BY                ') THEN
                BEGIN
                  INDIC:=2;  LNK:=2;   GOTO 800;
                END;
                IF TOKEN='WITH                 ' THEN
                 BEGIN WITH_BLOCK_PROCESSOR; GOTO 801; END;
                IF TOKEN='CASE                 ' THEN
                BEGIN BODY_PROCESSOR; GOTO 801; END;
              IF  DOT  THEN
              BEGIN
                 WC:=0;
                 WHILE  (DOT)  DO
                 BEGIN    DOT:=FALSE;
                          STRUC_VAR_HANDLING;
                          WC:=WC+1;  CODE:=3;   GET_TOKEN;
                 END;
                 CHNG_NL_VAR;
                 TWS:=TWS-WC;    GOTO 800;
              END;
              CHNG_NL_VAR;
              IF NOT(CHANGABLE) THEN
```

```
                 LOOKUP_PROC_NAMES(PROC_NAME_FOUND);
               800::
               CODE:=3;    GET_TOKEN;
               801::
            END;
            BACKTRACK_NLVAR;
            INDIC:=2;  LNK:=3;
        END;
(*************************************)
(*       WITH_BLOCK_PROCESSOR       *)
(*                                   *)
(*  ALL ITS ACTIONS ARE EXACTLY     *)
(*  SAME AS THOSE OF BODY           *)
(*  PROCESSOR AND REC BLOCK         *)
(*  PROCESSOR. THIS PROCESSOR IS    *)
(*  DESIGNED TO TAKE CARE OF ALL    *)
(*  POSSIBLE FORMS OF WITH STMTS.   *)
(*  EXAMPLE:                        *)
(*          WITH XX,XXX,... DO      *)
(*          BEGIN                   *)
(*             WITH X,XXXX,... DO   *)
(*                WITH XXXXX DO     *)
(*                     .........    *)
(*  ALL THESE NAMES X,XX,XXX,....   *)
(*  WILL BE STORED IN WITH STACK    *)
(*  AS THEY APPEAR, AND WILL BE     *)
(*  POPED OFF STACK AS THE BLOCK    *)
(*  IS ENDED......................  *)
(*************************************)
PROCEDURE WITH BLOCK PROCESSOR;
  VAR PROC_NAME FOUND:BOOLEAN;
      LAST_TWS,PIVOTBL,L,I,J:INTEGER;
                         FOUND:BOOLEAN;
   BEGIN
     LAST_TWS:=TWS;
     940:;
     CODE:=3;  GET_TOKEN;
     WHILE (TOKEN # 'DO                       ') DO
     BEGIN
          STRUC_VAR_HANDLING; CODE:=3; GET_TOKEN;
     END;
     CODE:=3; GET TOKEN;
     IF TOKEN='WITH                       ' THEN GOTO 940;
     IF TOKEN='BEGIN                      ' THEN
     BEGIN BODY PROCESSOR; GOTO 944; END;
     IF TOKEN='CASE                       ' THEN
     BEGIN BODY PROCESSOR; GOTO 944; END;
     IF TOKEN='ENSURE                     ' THEN
     BEGIN
       INDIC:=2; LNK:=1;
       REC_BLOCK_PROCESSOR; POP_STACK;


     END;
     REPEAT
        IF TOKEN='BEGIN                      ' THEN
        BEGIN  BODY PROCESSOR; GOTO 941; END;
        IF TOKEN='ENSURE                     ' THEN
        BEGIN
          INDIC:=2; LNK:=1;
          REC_BLOCK_PROCESSOR; POP_STACK;
        END;
        IF TOKEN='WITH                       ' THEN WITH_BLOCK_PROCESSOR;
        CHNG NL VAR;
        IF NOT(CHANGABLE) THEN LOOKUP_PROC_NAMES(PROC_NAME_FOUND);
        CODE:=3;  GET_TOKEN;
        941::
      UNTIL(SEMI_COLON)OR(PREV_TOKEN='END                    ');
      944::
      TWS:=LAST_TWS;
   END;
PROCEDURE  FILL END DCL PTR;
     (*  PROCEDURES ARE ALWAYS DECLARED IN  *)
     (*  DECLARATION AREA OF OTHER          *)
     (*  PROCEDURES. THE END OF DECLARATION *)
     (*  OF ANY PROCEDURE THEREFORE IS THE  *)
     (*  LAST DECLARATION BEFORE ITS MAIN   *)
     (*  BLOCK. THIS POINT MUST BE SAVED    *)
     (*  IN PROC/RB TREE FOR EACH PROCEDURE *)
     (*  TO BE USED LATER ON. WHEN A        *)
     (*  RECOVERY BLOCK HAS BEEN TRANSLATED *)
     (*  TO A PROCEDURE, IT MUST BE INSERTED*)
     (*  AT THE END OF DECLARATION OF ITS   *)
     (*  FATHER'S DECLARATION AREA......... *)
     VAR I,J:INTEGER;
     BEGIN
         PROC_NAME:=STACK[PRED(TOP_OF_STACK)].NAME;
         J:=HASH(PROC_NAME);
         I:=HASH_PTRS[J];
         WHILE(PROC_NAME # HASH_TABLE[I].PR_NAME) DO
         BEGIN
            REPEAT    J:=(J+1) MOD HASH_PTRS_LENGTH;
            UNTIL     (HASH_PTRS[J] # 0);
            I:=HASH_PTRS[J];
         END;
         J:=I;
                J:=HASH TABLE[J].PTR TO PROC TREE;
         PROC_RB_TREE[J].PTR_TO_END_DCL:=LINE_NO;
     END;
(*********************************************)
(*                                           *)
(*    MAIN BODY OF PROC_DCL_PROCESSOR.....   *)
(*                                           *)
(*********************************************)
```

```
 HEGIN
    IF FIRST_CALL THEN
    BEGIN
          CODE:=1;  GET_TOKEN;
    END;
    CREATE_PROC_NODE;
    INSERT_HASH_TABLE;
    PUSH_STACK;
    VAR_DCL_PROCESSOR;
    WHILE( (TOKEN='PROCEDURE                   ') OR
          (TOKEN='FUNCTION                    ')) DO
    BEGIN
       PRB:='PR';
       CODE:=4;    GET_TOKEN;
       LAST_PROC:=PROC_NAME;
       PROC_DCL_PROCESSOR;
    END;
    FILL_END_DCL_PTR;
    IF TOKEN # 'BEGIN                       ' THEN
    BEGIN
       WRITELN('**** ERROR AT STMT NO.',STMT_NO:4);
       WRITELN('**** BEGIN EXPECTED, BUT ENCOUNTERED ',TOKEN);
    END;
    CODE:=2;  INDIC:=2;  LNK:=4;  GET_TOKEN;
    INDIC:=1;  TWS:=1;
    BODY_PROCESSOR;
    BACKTRACK_NLVAR;
    POP_STACKT
    9007;
 END;
 PROCEDURE COPY_BACK;
    VAR I:INTEGER;
    BEGIN
      FOR I:=1 TO NEXT_NL-1 DO
      BEGIN NL_VAR_TABLE[I].INDICATOR:=NL_BUFER[I].INDICATOR;
         NL_VAR_TABLE[I].PTR_TO_VDTAB:=
         NL_BUFER[I].PTR_TO_VDTAB;
      END;
      NEXT_NL_VAR:=NEXT_NL;
    END;

(*****************************************)
(*                                       *)
(*         MAIN BODY OF  PASS_1          *)
(*                                       *)
(*****************************************)
BEGIN
     INITIALIZE_PASS1;
     PROC_DCL_PROCESSOR;
     COPY_BACK;
     PRINT_RESULT;


END;
  PROCEDURE SAVE_LINE(VAR FILEN:TEXT_FILE;VAR BUFER :TEXT_LINE;
                                          VAR LINE_NO:INTEGER);
  FORWARD;
  PROCEDURE READ_LINE(VAR FILEN:TEXT_FILE;VAR BUFFER:TEXT_LINE;
                                          VAR LINE_NO:INTEGER);
     VAR I,J:INTEGER;
     BEGIN I:=1;
           WHILE (NOT EOLN(FILEN)) DO
           BEGIN READ(FILEN,BUFFER[I]); I:=I+1; END;
           READLN(FILEN);
           FOR J:=I TO LINE_LENGTH DO BUFFER[J]:=' ';
           LINE_NO:=LINE_NO+1;   TXT_PTR:=1;
     END;

  PROCEDURE SAVE_LINE;
     BEGIN WRITELN(FILEN,BUFER);
           LINE_NO:=LINE_NO+1;   COL:=1;
           WHILE((BUFER[COL]=' ')AND(COL<LINE_LENGTH))DO COL:=COL+1;
           IF COL=LINE_LENGTH THEN COL:=1;
     END;
   (*****************************************************)
   (*                     PASS TWO                      *)
   (*                                                   *)
   (*   PASS 2 GOES THRU THE INPUT PROGRAM ONCE MORE AND *)
   (*   USING THE CONTENTS OF SEVERAL TABLES GENERATED BY *)
   (*   PASS 1,WILL DO THE FOLLOWING:                    *)
   (*                                                   *)
   (*   1.   ANY FIRST LEVEL RECOVERY BLOCK WILL BE TRANS *)
   (*        LATED INTO A PROCEDURE AND COPIED INTO A FILE *)
   (*        CALLED INCREMENT TEXT.                      *)
   (*   2.   THE RECOVERY BLOCK IN THE ORIGINAL TEXT WILL *)
   (*        BE IGNORED AND REPLACED BY A CALL TO JUST    *)
   (*        TRANSLATED RECOVERY BLOCK.                  *)
   (*   3.   THE FIRST TWO STEPS WILL BE PERFORMED FOR ALL *)
   (*        FIRST LEVEL RECOVERY BLOCKS IN THE TEXT INPUT *)
   (*        FROM PASS 1.                                *)
   (*   4.   IF THERE ARE NESTED REC.BLOCKS,PASS 2 WILL BE *)
   (*        CALLED AGAIN THIS TIME ITS ARGUMENT IS THE   *)
   (*        INCREMENT TEXT,THE OUTPUT FROM PASS 2 ITSELF. *)
   (*   5.   PASS 2 WILL BE CALLED AS LONG AS THERE ARE   *)
   (*        MORE LEVELS OF RECOVERY BLOCKS.             *)
   (*   6.   AT THE VERY END,ALL THESE TEXTS WILL BE MERGED *)
   (*        TOGETHER AND THAT'S WHAT CAN BE ACCEPTED BY   *)
   (*        A REGULAR PASCAL COMPILER.                  *)
   (*                                                   *)
   (*   IN GENERAL PASS 2 HAS THE FOLLOWING FORM:        *)
   (*                                                   *)
   (*                 FROM PASS_1                       *)
   (*                      .                            *)
   (*                      .                            *)
(*                          .                                    *)
(*                                                               *)
(*                    I------------I                             *)
(*                    I   PASS_2   I                             *)
(*                    I------------I                             *)
(*                        .  .  .                                *)
(*                      .         .                              *)
(*                    .             .                            *)
(*                  .                 .                          *)
(*          TEXT1                INCREMENT_TEXT1                 *)
(*                                     .                         *)
(*                                     .                         *)
(*                                     .                         *)
(*                                     .                         *)
(*                             I------------I                    *)
(*                             I   PASS_2   I                    *)
(*                             I------------I                    *)
(*                                 .  .  .                       *)
(*                               .         .                     *)
(*                             .             .                   *)
(*                           .                 .                 *)
(*                       TEXT2          INCREMENT_TEXT2          *)
(*                                          .                    *)
(*                                          .                    *)
(*                                          .                    *)
(*                                          .                    *)
(*  TEXT1,TEXT2,...AND THE LAST INCREMENT_TEXT TO BE             *)
(*  MERGED.                                                      *)
(*                                                               *)
(***************************************************************)

PROCEDURE PASS_2(VAR TEXT1,TEXT2,TEXT3:TEXT_FILE);
   VAR
        I,J,K,N:INTEGER;
        BUF,LASTLINE:TEXT_LINE;
        NODE,NODEPTR,SEARCH_DIRECTION:INTEGER;
        OLDPTR:INTEGER;
        ELDER_BROTHER,ELD_PTR:INTEGER;
        PRO:STRING9;
        RB_TO_PROC:ARRAY[1..NO_OF_PROC] OF INTEGER;
   PROCEDURE PUT_LINE(VAR FILEN:TEXT_FILE; VAR BUFER:TEXT_LINE;
                                           VAR LINE_NO:INTEGER);
      VAR I:INTEGER;
      (*  A GENERAL PROCEDURE TO WRITE A TEXT LINE   *)
      (*  ON A FILE.'COL' IS A VARIABLE POINTING TO  *)
      (*  THE STARTING OF EACH LINE JUST WRITTEN ON  *)
      (*  ON FILE. IT IS USED FOR THE PURPOSE OF     *)
      (*  INDENTATION............................... *)
      BEGIN  WRITELN(FILEN,BUFER);
             LINE_NO:=LINE_NO+1;  I:=1;
             WHILE((BUFER[I]=' ') AND (I<LINE_LENGTH)) DO


             I:=I+1;  IF I< LINE_LENGTH-1 THEN COL:=I;
      END;
   PROCEDURE GET_LINE(VAR FILEN:TEXT_FILE; VAR BUFFER:TEXT_LINE;
                                           VAR LINE_NO:INTEGER);
      VAR I,J:INTEGER;
          BLANK,COMENT:BOOLEAN;
      (*  A GENERAL  PROCEDURE TO READ A TEXT LINE   *)
      (*  FROM A FILE. COMMENTS AND BLANK LINES      *)
      (*  WILL BE TAKEN CARE HERE................... *)
      BEGIN I:=1;
            WHILE(NOT EOLN(FILEN))DO
            BEGIN READ(FILEN,BUFFER[I]);  I:=I+1;  END;
            READLN(FILEN);
            FOR J:=I TO LINE_LENGTH DO BUFFER[J]:=' ';
            LINE_NO:=LINE_NO+1;    TXT_PTR:=1;
            IF INDIC=1 THEN GOTO 771;
            BLANK:=TRUE;    COMENT:=FALSE;
            FOR J:=1 TO LINE_LENGTH-1 DO
            BEGIN IF BUFFER[J]#' ' THEN
                  BEGIN BLANK:=FALSE;
                        IF((BUFFER[J]='(') AND
                           (BUFFER[J+1]='*')) THEN
                        BEGIN  COMENT:=TRUE;  GOTO 770;  END
                        ELSE   GOTO 771;
                  END;
            END;
            770::
            IF (BLANK  OR  COMENT) THEN
            BEGIN  IF INDIC=2 THEN PUT_LINE(TEXT2,BUFFER,PTR2)
                              ELSE PUT_LINE(TEXT3,BUFFER,PTR3);
                  GET_LINE(FILEN,BUFFER,LINE_NO);
            END;
            771::
      END;
   PROCEDURE FCOPY(VAR TEXT1,TEXT3:TEXT_FILE; VAR BUF:TEXT_LINE;
                                         VAR PTR1,PTR3,N:INTEGER);
      BEGIN
         (*  COPY A PORTION OF ANY FILE  *)
         (*  INTO ANY OTHER FILE.        *)
         WHILE (PTR1<N) DO
         BEGIN    INDIC:=1;  GET_LINE(TEXT1,BUF,PTR1);
                  PUT_LINE(TEXT3,BUF,PTR3);
         END;
      END;

   PROCEDURE FILL_ADR_FIELD(VAR BUFER:TEXT_LINE;VAR J,K:INTEGER);
   FORWARD;
   PROCEDURE  GENERATE_CALL(VAR BUF:TEXT_LINE;VAR J:INTEGER);
   FORWARD;

   PROCEDURE CREATE_LINKAGE;
   (*  CREATE A 'SLINKAGE ADR' STAEMENT,WHERE   *)
   (*  ADR POINTS TO INCREMENT TEXT. IT WILL    *)
   (*  USED IN MERGING THE TEXTS............... *)
   BEGIN
         K:=PROC_RB_TREE[NODEPTR].FATHER;
         N:=PROC_RB_TREE[K].PTR_TO_END_DCL-1; J:=10;
         FILL_ADR_FIELD(LASTLINE,J,N);
         PROC_RB_TREE[K].PTR_TO_END_DCL:=PTR2;
         GENERATE_CALL(BUFER,NODEPTR);
         PUT_LINE(TEXT3,BUFER,PTR3);
   END;

FUNCTION YOUNGER(VAR NODEPTR:INTEGER):BOOLEAN;
   VAR I:INTEGER;
   (*  THIS FUNCTION RETURNS TRUE IF   *)
   (*  THE PROCEDURE OR RECOVERY BLCK  *)
   (*  POINTED TO BY NODEPTR HAS ANY   *)
   (*  YOUNGER BROTHER. OTHERWISE IT   *)
   (*  RETURNS FALSE.................  *)
   BEGIN
     I:=NODEPTR;  YOUNGER:=FALSE;
     WHILE (PROC_RB_TREE[I].YOUNGER_BROTHER #0) DO
     BEGIN
       I:=PROC_RB_TREE[I].YOUNGER_BROTHER;
       IF PROC_RB_TREE[I].INDICATOR ='RB' THEN YOUNGER:=TRUE;
     END;
   END;

PROCEDURE  FIND_ELDER_BROTHER;
   VAR I,J:INTEGER;
   (*  RETURNS THE ELDER BROTHER OF   *)
   (*  THE RECOVERY BLOCK OR          *)
   (*  PROCEDURE POINTED TO BY        *)
   (*  NODEPTR IF ANY. OTHERWISE      *)
   (*  RETURNS NULL.(NULL=1)          *)
   BEGIN
         ELDER_BROTHER:=1;
         I:=PROC_RB_TREE[NODEPTR].FATHER;
         I:=PROC_RB_TREE[I].FIRST_SON;
         IF I=NODEPTR THEN GOTO 150;
         WHILE (I #NODEPTR) DO
         BEGIN J:=I;
               I:=PROC_RB_TREE[I].YOUNGER_BROTHER;
         END;
         IF PROC_RB_TREE[J].INDICATOR='PR' THEN
                                      ELDER_BROTHER:=1
                             ELSE  ELDER_BROTHER:=2;
   150:;
   END;

PROCEDURE  CONVERT_DECIMAL(VAR BUFER:TEXT_LINE;


                             VAR I,J,K:INTEGER);
   VAR M:INTEGER;
   (*  A PORTION OF THE TEXT LINE IN BUFER   *)
   (*  STARTING FROM J TH POSITION AND       *)
   (*  ENDING AT I TH POSITION IS CONVERTED  *)
   (*  TO A DECIMAL NUMBER.                  *)
   (*       STRING =========> NUMBER         *)
   BEGIN
         K:=0;  M:=1;
         REPEAT K:=K+(ORD(BUFER[J])-48)*M;
                J:=J-1;  M:=M*10;
         UNTIL  (J<I);
   END;

PROCEDURE  FILL_ADR_FIELD;
   VAR M,I,L:INTEGER;
   (*  A NUMBER K  WILL BE CONVERTED TO A   *)
   (*  STRING AND WILL BE COPIED INTO THE   *)
   (*  TEXT LINE LOCATED AT BUFER STARTING  *)
   (*  FROM J TH POSITION.                  *)
   (*       NUMBER ========> STRING         *)
   BEGIN
         M:=1000;  L:=K;
         REPEAT  I:=K DIV M;  K:=K-I*M;
                 BUFER[J]:=CHR(ORD(I)+48);
                 J:=J+1;  M:=M DIV 10;
         UNTIL   (M=0);
         K:=L;
   END;

PROCEDURE GENERATE_CALL;
   VAR NAME:IDENTIFIER;
          I:INTEGER;
   BEGIN
         (*  GENERATE A CALL TO THE GENERATED  *)
         (*  PROCEDURE CORRESPONDING TO THE    *)
         (*  RECOVERY BLOCK..................  *)
         NAME:=PROC_RB_TREE[J].PR_RB_NAME;
         FOR I:=1 TO LINE_LENGTH DO BUF[I]:=' ';
         FOR I:=1 TO ID_LENGTH DO BUF[COL+I-1]:=NAME[I];
         IF((NOT SEMICOLON) AND (LEE=LASTELSERROR))
         THEN SEMICOLON:=TRUE  ELSE  BUF[COL+I-1]:=';';
   END;

PROCEDURE  DCL_FAULTFLAG;
   VAR I,J:INTEGER;
      NAME:STRING20;
   (*  GENERATE A DECLARATION STAEMENT FOR  *)
   (*  BOOLEAN VARIABLE CALLED 'FAULTFLAG'  *)
   (*  AT THE END OF DECLARATION AREA OF    *)
   (*  THE MAIN PROCEDURE.................. *)
```

```
    BEGIN
        I:=PROC_RB_TREE[NODEPTR].FIRST_SON;
        IF PROC_RB_TREE[I].INDICATOR='PR'
        THEN  I:=PROC_RB_TREE[I].ENTRY
        ELSE I:=PROC_RB_TREE[NODEPTR].PTR_TO_END_DCL+1;
        FCOPY(TEXT1,TEXT3,BUFER,PTR1,PTR3,I);
        I:=COL;
        FOR J:=1 TO LINE_LENGTH DO BUFER[J]:=' ';
        NAME:='FAULTFLAG:BOOLEAN;   ';
        FOR J:=1 TO 20 DO  BUFER[I+J-1]:=NAME[J];
        PUT_LINE(TEXT3,BUFER,PTR3);
    END;
PROCEDURE  NEW_NODE(VAR SEARCH_DIRECTION:INTEGER);
    VAR J,I:INTEGER;
    (*  TRAVELLING ON PROC/RB TREE  *)
    (*  IS DEPTH FIRST SEARCH .     *)
    BEGIN
      CASE  SEARCH_DIRECTION  OF
      1:  BEGIN
            (*     DIRECTION <====   SON     *)
            I:=PROC_RB_TREE[NODEPTR].FIRST_SON;
            J:=NODEPTR;
            IF  I#0  THEN
            BEGIN
               (*  AS FAR AS NODE HAS BEEN VISITED  *)
               (*  GO ON TO THE NEAT NODE.          *)
               WHILE (PROC_RB_TREE[I].MARK='V') DO
               BEGIN J:=I; I:=PROC_RB_TREE[I].FIRST_SON;
                     IF I=0 THEN  GOTO 100;
               END;
               NODEPTR:=I;
               IF PROC_RB_TREE[I].INDICATOR='PR'
                  THEN NODE:=1  ELSE NODE:=3;
                  GOTO 200;
            END;
            100:;
            NODE:=2;  NODEPTR:=J;
            200:;
          END;

      2:  BEGIN
            (*     DIRECTION <======= BROTHER    *)
            I:=PROC_RB_TREE[NODEPTR].YOUNGER_BROTHER;
            WHILE (PROC_RB_TREE[I].MARK='V') DO
                  I:=PROC_RB_TREE[I].YOUNGER_BROTHER;
            NODEPTR:=I;
            IF PROC_RB_TREE[I].INDICATOR='PR'
               THEN NODE:=1  ELSE NODE:=3;
          END;

      3:  BEGIN


             (*  RESERVED  *)
          END
      END;
    END;

PROCEDURE  INITIATE_PASS2;
    VAR I:INTEGER;
    BEGIN
        RESET(TEXT1);  REWRITE(TEXT3);
        IF NEED_MERGE THEN RESET(TEXT2)
                     ELSE REWRITE(TEXT2);
        PRO:='$LINKAGE ';
        FOR I:=1 TO LINE_LENGTH DO LASTLINE[I]:=' ';
        FOR I:=1 TO 9 DO  LASTLINE[I]:=PRO[I];
        PTR1:=1; PTR2:=1; PTR3:=1;
        SEARCH_DIRECTION:=1;
        NODE:=1; NODEPTR:=1; NEED_MORE_PASS:=FALSE; LIN:=1;
        COLMAKER:=COLMAKER+1;
    END;

PROCEDURE  GENERATE_PROC(VAR BUF:TEXT_LINE);
    (*  GENERATE PROCEDURE HEADING FOR THE  *)
    (*  PROCEDURE CORRESPONDING TO THE      *)
    (*  RECOVERY BLOCK..................... *)

    VAR I,J:INTEGER;
    BEGIN
        PRO:='PROCEDURE';
        COL:=COLMAKER+5;
        FOR I:=1 TO LINE_LENGTH DO BUF[I]:=' ';
        RB_NAME:=PROC_RB_TREE[NODEPTR].FR_RB_NAME;
        PROC_RB_TREE[NODEPTR].ENTRY:=PTR2;
        FOR I:=0 TO 8 DO BUF[COL+I]:=PRO[I+1];
        FOR J:=1 TO ID_LENGTH DO BUF[COL+I+J]:=RB_NAME[J];
        BUF[COL+I+J]:=';';
    END;

FUNCTION  HASH(VAR NAME:IDENTIFIER):INTEGER;
    VAR I,J,TEN,SUM:INTEGER;
    BEGIN
        I:=1; TEN:=1; SUM:=0;
        WHILE (I<=ID_LENGTH) DO
        BEGIN  J:=ORD(NAME[I]);
               IF J>64 THEN J:=J-44 ELSE J:=J-32;
               SUM:=SUM+J*TEN; TEN:=TEN*10; I:=I+1;
        END;
        HASH:=SUM MOD HASH_PTRS_LENGTH;
    END;

PROCEDURE  NAME_CREATION(VAR NAME:IDENTIFIER);
    (*  A GENERAL PROCEDURE TO CREATE NEXT  *)
    (*   NAME FOR 'SAVE' OR 'RESTORE' OR      *)
    (*   'VALIDATION FUNCTION'. NAMES HAVE    *)
    (*   THE FOLLOWING FORM:                  *)
    (*                                        *)
    (*     SAVE -----> SA000001,SA000002,.... *)
    (*     RESTORE --> RS000001,RS000002,.... *)
    (*     FUNCTION -> VT000001,VT000002,.... *)
    (*                                        *)
    VAR I:INTEGER;
    BEGIN
          I:=ORD(NAME[ID_LENGTH]);
          IF I=57 THEN
          BEGIN NAME[ID_LENGTH]:='0';
                NAME[ID_LENGTH-1]:=
                CHR(SUCC(ORD(NAME[ID_LENGTH-1])));
          END
          ELSE  NAME[ID_LENGTH]:=
                CHR(SUCC(ORD(NAME[ID_LENGTH])));
    END;


PROCEDURE BV_NAME(VAR NAME:IDENTIFIER; VAR I:INTEGER);
    VAR J,K,S:INTEGER;
    (*   CREATE A BACKUP VARIABLE NAME FOR   *)
    (*   THE IDENTIFIER CHARACTERIZED BY     *)
    (*   POINTER I POINTING TO NLVAR_LIST.   *)
    (*   ANY BACKUP VARIABLE NAME HAS THE    *)
    (*   FOLLOWING FORMAT:                   *)
    (*   BV000001,BV000002,BV000003,........ *)
    (*   THEY ARE NOT CREATED IN SEQUENCE.   *)
    BEGIN
       S:=I;
       FOR K:=0 TO ID_LENGTH-3 DO
       BEGIN J:=I MOD 10; I:=I DIV 10;
             NAME[ID_LENGTH-K]:=CHR(ORD(J)+48);
       END;
       I:=S;
    END;

PROCEDURE OBJBLK_TRANSLATION(VAR NODEPTR:INTEGER); FORWARD;
PROCEDURE VALIDATION_FUNCTION;  FORWARD;

PROCEDURE RB_TRANSLATION(VAR NODEPTR:INTEGER);
    VAR PTRS:ARRAY[0..PTRS_LENGTH] OF INTEGER;
        I:INTEGER;

    PROCEDURE  FIND_NLOC_VAR(VAR NODEPTR:INTEGER);
       (*  I: LO POINTER TO NL_VAR_TABLE   *)
       (*  J: HI POINTER TO NL_VAR_TABLE   *)
       (*  BOTH POINTING TO THAT PORTION   *)
       (*  OF THE TABLE WHICH IS ASSIGNED  *)


       (*  TO THE PROCEDURE OR REC.BLOCK   *)
       (*  POINTED TO BY NODEPTR.........  *)
       (*  IF A PROCEDURE OR REC.BLOCK IS  *)
       (*  NONLOCAL TO THIS PROCEDURE,     *)
       (*  IT WILL CALL ITSELF RECURSIVELY *)
       (*  TO FIND ALL NONLOCAL VARIABLES  *)
       (*  CHANGABLE.                      *)

       VAR I,J,K,L,M:INTEGER;
              CODE:STRING2;
       BEGIN
          I:=PROC_RB_TREE[NODEPTR].PTR_TO_NLVAR;
          J:=PROC_RB_TREE[NODEPTR].NO_NLVAR_USED;
          J:=J+I;
          WHILE (I<J) DO
          BEGIN
             CODE:=NL_VAR_TABLE[I].INDICATOR;
             K:=NL_VAR_TABLE[I].PTR_TO_VOTAB;
             IF  (CODE='PR') OR (CODE='RB')
             THEN FIND_NLOC_VAR(K)
             ELSE IF CODE='ID'
               THEN BEGIN
                       FOR L:=1 TO NO_NLOC-1 DO
                       BEGIN
                         M:=NLVAR_LIST[L];
                         IF (M#1) AND
                            (NL_VAR_TABLE[M].INDICATOR#'CD')
                         THEN BEGIN
                                IF NL_VAR_TABLE[M-1].INDICATOR
                                   #'CD' THEN
                                IF NL_VAR_TABLE[M].PTR_TO_VOTAB
                                  =NL_VAR_TABLE[I].PTR_TO_VOTAB
                                  THEN GOTO 589;
                              END;
                       END;
                       NLVAR_LIST[NO_NLOC]:=I;
                       NO_NLOC:=NO_NLOC+1;
                       589:;
                    END
               ELSE BEGIN
                       NLVAR_LIST[NO_NLOC]:=I;
                       NO_NLOC:=NO_NLOC+1;
                       WHILE (NL_VAR_TABLE[I].INDICATOR='CD') DO
                       I:=I+1;
                    END;
             I:=I+1;
          END;
       END;

    PROCEDURE NL_ID_TYPE( VAR I:INTEGER; VAR TYPP:IDENTIFIER);
       VAR J,K,MTL:INTEGER;
```

```
          TN:IDENTIFIER;
    (*  FIND THE TYPE OF NONLOCAL VARIABLE  *)
    (*  POINTED TO BY THE INDEX I.          *)
    BEGIN
       IF NL_VAR_TABLE[I].INDICATOR='ID' THEN
       BEGIN
          (*    SIMPLE  VARIABLE    *)
          K:=NL_VAR_TABLE[I].PTR_TO_VOTAB;
          J:=VAR_DCL_TABLE[K].TCODE;
          IF VAR_DCL_TABLE[K].INDIC=1
             THEN TYPP:=SIMPLE_TABLE[J]
             ELSE GIVE_TYPE(TYPP,J);
       END ELSE
       BEGIN
          (*    STRUCTURE  VARIABLE        *)
          (*  IN THIS CASE STOP LOOKING    *)
          (*  FOR TYPE IF THE TYPE IS      *)
          (*  ARRAY.                       *)
          M:=I;
          WHILE(NL_VAR_TABLE[M].INDICATOR='CD') DO M:=M+1;
          K:=NL_VAR_TABLE[M].PTR_TO_VOTAB;
          IF (VAR_DCL_TABLE[K].INDIC=3) OR
             (VAR_DCL_TABLE[K].INDIC=4) THEN
          BEGIN L:=VAR_DCL_TABLE[K].TTCODE;
                TYPP:=SIMPCE_TABLE[L];  GOTO 735;
          END;
          K:=VAR_DCL_TABLE[K].TCODE;
          IF STRUCTURE_TABLE[K].IND1='A' THEN
          BEGIN
             TYPP:=STRUCTURE_TABLE[K].NAME;  GOTO 735;
          END
          ELSE GIVE_TYPE(TYPP,K);
          M:=M-1;
          WHILE (M>=I) DO
          BEGIN
             K:=NL_VAR_TABLE[M].PTR_TO_VOTAB;
             GIVE_TYPET(TYPP,K);  M:=M-I;
          END;
       END;
       735:;
    END;

PROCEDURE NL_ID_NAME( VAR I:INTEGER; VAR NAME:IDENTIFIER;
                      VAR TEMP:STRING45; VAR CD:BOOLEAN);
    VAR L,J,K,M:INTEGER;
          TN:IDENTIFIER;
    (*  GIVE THE NAME OF NONLOCAL VARIABLE  *)
    BEGIN
       IF NL_VAR_TABLE[I].INDICATOR='ID' THEN
       BEGIN
          (*    SIMPLE VARIABLE    *)


          K:=NL_VAR_TABLE[I].PTR_TO_VOTAB;
          NAME:=VAR_DCL_TABLE[K].VNAME;
          CD:=FALSE;
       END ELSE
       BEGIN
          (*    STRUCTURE  VARIABLE      *)
          (*  IN THIS CASE NAME OF THE   *)
          (*  NONLOCAL VARIABLE HAS      *)
          (*  FOLLOWING FORM:            *)
          (*  XX.XXX.X..... THEN STOP    *)
          (*  BUILDING THIS NAME WHEN    *)
          (*  THE TYPE OF LAST APPENDED*)
          (*  COMPONENT IS ARRAY.......*)
          M:=I;  CD:=TRUE;
          WHILE(NL_VAR_TABLE[M].INDICATOR='CD')DO M:=M+1;
          FOR L:=1 TO 45 DO TEMP[L]:=' ';
          L:=0;  K:=NL_VAR_TABLE[M].PTR_TO_VOTAB;
          TN:=VAR_DCL_TABLE[K].VNAME;
          FOR J:=I TO ID_LENGTH DO
          BEGIN
             IF TN[J]#' ' THEN
             BEGIN  L:=L+1; TEMP[L]:=TN[J]; END;
          END;
          L:=L+1;  TEMP[L]:='.';
          IF (VAR_DCL_TABLE[K].INDIC=3) OR
             (VAR_DCL_TABLE[K].INDIC=4) THEN GOTO 737;
          K:=VAR_DCL_TABLE[K].TCODE;
          IF STRUCTURE_TABLE[K].IND1='A' THEN GOTO 737;
          M:=M-1;
          WHILE (M>=I) DO
          BEGIN
             K:=NL_VAR_TABLE[M].PTR_TO_VOTAB;
             TN:=STRUCTURE_TABLE[K].NAME;
             FOR J:=1 TO ID_LENGTH DO
             BEGIN
                IF TN[J]#' ' THEN
                BEGIN  L:=L+1; TEMP[L]:=TN[J];  END;
             END;
             L:=L+1;    TEMP[L]:='.';
             M:=M-1;
          END;
          737:;
          TEMP[L]:=' ';
          FOR J:=1 TO ID_LENGTH DO NAME[J]:=' ';
          J:=1;
          WHILE ((J<=ID_LENGTH) AND (TEMP[J]#'.')) DO
          BEGIN
             NAME[J]:=TEMP[J];  J:=J+1;
          END;
       END;
       736:;
```

```
   END;

PROCEDURE  DCL_BKUP_VAR;
   (*  FIND THE NAME AND THE TYPE OF   *)
   (*  THE NONLOCAL VARIABLE AND THEN  *)
   (*  DECLARE A BACKUP VARIABLE FOR   *)
   (*  THAT.  EXAMPLE:                 *)
   (*  IF IDENTIFIER NAME IS ID AND    *)
   (*  IDENTIFIER TYPE IS IDTYPE THEN: *)
   (*       VAR  BV000001ID:IDTYPE     *)
   (*  IS WHAT WE ARE LOOKING FOR.     *)
   VAR I,J,K:INTEGER;
       TYPP,NAME:IDENTIFIER;
           TEMP:STRING45;
             CD:BOOLEAN;
   BEGIN
       FOR I:=1 TO LINE_LENGTH DO BUFER[I]:=' ';
       BUFER[COL+3]:='V'; BUFER[COL+4]:='A';
       BUFER[COL+5]:='R';
       PUT_LINE(TEXT2,BUFER,PTR2);
       FOR I:=1 TO NO_NLOC-1  DO
       BEGIN
          FOR K:=1 TO LINE_LENGTH DO BUFER[K]:=' ';
          K:=NLVAR_LIST[I]; NAME:='BV000000';
          BV_NAME(NAME,I);
          FOR J:=1 TO ID_LENGTH DO BUFER[COL+J+3]:=NAME[J];
          J:=COL+J+3;
          NL_ID_NAME(K,NAME,TEMP,CD);
          NL_ID_TYPE(K,TYPP);
          FOR K:=1 TO ID_LENGTH DO BUFER[J+K-1]:=NAME[K];
          BUFER[K+J-1]:=':';  J:=K+J;
          FOR K:=1 TO ID_LENGTH DO BUFER[K+J]:=TYPP[K];
          BUFER[K+J]:=';';
          PUT_LINE(TEXT2,BUFER,PTR2);
          COL:=COL-4;
       END;
    END;

PROCEDURE PROC_HEADING(VAR PSVR:IDENTIFIER);
   (*  GENERATE PROCEDURE HEADING FOR  *)
   (*  SAVE AND RESTORE PROCEDURES.    *)
   (*  PSVR  CARRIES THE NAME OF THESE *)
   (*  PROCEDURES..................... *)
   VAR I,J:INTEGER;
       CMNT:STRING16;
   BEGIN
        PRO:='PROCEDURE';
        FOR I:=1 TO LINE_LENGTH DO BUFER[I]:=' ';
        FOR I:=0 TO 8 DO BUFER[COL+I]:=PRO[I+1];


        BUFER[COL+I]:=' '; NAME_CREATION(PSVR);
        FOR J:=1 TO ID_LENGTH DO
        BUFER[COL+I+J+1]:=PSVR[J];
        BUFER[COL+I+J+1]:=';';
        IF CODE=1 THEN  CMNT:='   (* SAVE *)   ';
                  ELSE  CMNT:='   (* RESTORE *)';
        I:=COL+I+J+2;
        FOR J:=1 TO 16 DO BUFER[I+J]:=CMNT[J];
        PUT_LINE(TEXT2,BUFER,PTR2);
        PRO:='BEGIN    ';
        FOR I:=1 TO LINE_LENGTH DO BUFER[I]:=' ';
        FOR I:=0 TO 8 DO BUFER[COL+I+3]:=PRO[I+1];
        PUT_LINE(TEXT2,BUFER,PTR2);
   END;

PROCEDURE SAVE_VAR;
   (*  GENERATE STAEMENTS FOR THE BODY OF  *)
   (*  SAVE PROCEDURE. THE STAEMENTS MUST  *)
   (*  BE ABLE TO STORE THE ORIGINAL VALUE *)
   (*  OF NONLOCAL VARIABLES INTO THE BACK *)
   (*  UP VARIABLES....................... *)
   VAR I,J,K,L,M:INTEGER;
           NAME:IDENTIFIER;
             NAM:DOUBLE_ID;
               TE:STRING45;
               CD:BOOLEAN;
   BEGIN
      CODE:=1;  PROC_HEADING(PVS);
      FOR I:=1 TO NO_NLOC-1 DO
      BEGIN
         FOR J:=1 TO LINE_LENGTH DO BUFER[J]:=' ';
         NAME:='BV000000';  BV_NAME(NAME,I);
         FOR J:=1 TO ID_LENGTH DO NAM[J]:=NAME[J];
         K:=NLVAR_LIST[I];
         NL_ID_NAME(K,NAME,TE,CD);
         FOR J:=1 TO ID_LENGTH DO NAM[J+ID_LENGTH]:=NAME[J];
         FOR J:=1 TO ID_LENGTH*2 DO BUFER[COL+J+4]:=NAM[J];
         BUFER[COL+J+4]:=':';  BUFER[COL+J+5]:='=';
         J:=COL+J+6;
         IF CD THEN
         BEGIN
            FOR L:=1 TO 45 DO
            BEGIN
               IF TE[L]#' ' THEN
               BEGIN M:=L; BUFER[J+L-1]:=TE[L]; END;
            END;
            BUFER[J+M]:=';';
         END
         ELSE BEGIN
                 FOR L:=1 TO ID_LENGTH DO BUFER[J+L-1]:=
                                                         NAME[L];
                        BUFER[J+L]:=';';
                     END;
               PUT_LINE(TEXT2,BUFER,PTR2);
               COL:=COL-5;
           END;
           PRO:='END;       ';
           FOR K:=1 TO LINE_LENGTH DO BUFER[K]:=' ';
           FOR K:=0 TO 3 DO BUFER[COL+K]:=PRO[K+1];
           PUT_LINE(TEXT2,BUFER,PTR2);
           COL:=COL-3;
       END;

   PROCEDURE RESTORE_VAR;
       (*  GENERATE STAEMENTS FOR THE BODY OF   *)
       (*  RESTORE PROCEDURE. THESE STAEMENTS   *)
       (*  MUST BE ABLE TO STORE BACK THE       *)
       (*  ORIGINAL VALUE OF NONLOCAL VARIABLES*)
       (*  FROM BACKUP VARIABLES INTO THE VARS *)
       VAR I,J,K,M,L:INTEGER;
            NAME,NA:IDENTIFIER;
           NAM:DOUBLE_ID;
           TE:STRING45;
           CD:BOOLEAN;
       BEGIN
           CODE:=2;  PROC_HEADING(PRV);
           FOR I:=1 TO NO_NLOC-1 DO
           BEGIN
               FOR J:=1 TO LINE_LENGTH DO BUFER[J]:=' '; J:=COL+4;
               K:=NLVAR_LIST[I];
               NL_ID_NAME(K,NAME,TE,CD);
               IF CD THEN
               BEGIN
                   FOR M:=1 TO 45 DO
                   BEGIN
                       IF TE[M]#' ' THEN
                       BEGIN L:=M; BUFER[J+M-1]:=TE[M]; END;
                   END;
                   BUFER[J+L]:=':';  BUFER[J+L+1]:='=';
                   J:=J+L+2;
               END
               ELSE BEGIN
                         FOR M:=1 TO ID_LENGTH DO
                         BUFER[J+M-1]:=NAME[M];
                         BUFER[J+M-1]:=':'; BUFER[J+M]:='=';
                         J:=J+M+1;
                     END;
               NA:='BV000000';    BV_NAME(NA,I);
               FOR L:=1 TO ID_LENGTH DO NAM[L]:=NA[L];
               FOR L:=1 TO ID_LENGTH DO NAM[L+ID_LENGTH]:=NAME[L];


               FOR L:=1 TO ID_LENGTH*2 DO BUFER[L+J-1]:=NAM[L];
               BUFER[L+J-1]:=';';
               PUT_LINE(TEXT2,BUFER,PTR2);
               COL:=COL-4;
           END;
           PRO:='END;       ';
           FOR K:=1 TO LINE_LENGTH DO BUFER[K]:=' ';
           FOR K:=0 TO 3 DO BUFER[COL+K]:=PRO[K+1];
           PUT_LINE(TEXT2,BUFER,PTR2);
           COL:=COL-3;
       END;

    (*   MAIN BODY OF RB_TRANSLATION    *)
    BEGIN
         RB_TO_PROC[LIM]:=NODEPTR;  LIM:=LIM+1;
         NO_NLOC:=1;
         FOR I:=0 TO PTRS_LENGTH DO PTRS[I]:=0;
         GENERATE_PROC(BUFER);
         PUT_LINE(TEXT2,BUFER,PTR2);
         FIND_NLOC_VAR(NODEPTR);
         DCL_BKUP_VAR;
         SAVE_VAR;
         RESTORE_VAR;
         INDIC:=3;
         GET_LINE(TEXT1,BUFER,PTR1); GET_LINE(TEXT1,BUFER,PTR1);
         VALIDATION_FUNCTION;
         INDIC:=2;
         GET_LINE(TEXT1,TEMP_STMT,PTR1);
         GET_LINE(TEXT1,TEMP_STMT,PTR1);
         GET_LINE(TEXT1,TEMP_STMT,PTR1);
         OBJBLK_TRANSLATION(NODEPTR);
    END;

PROCEDURE VALIDATION_FUNCTION;
    VAR TEST,T:STRING47;
         TST:STRING50;
         I,J,KK,K,M:INTEGER;
         BLANK:BOOLEAN;
    PROCEDURE  SUBSTITUTE_PRIOR(VAR TEST:STRING47;
                                VAR K,KK:INTEGER);
       VAR P:STRING5;
            N,O,L,S,U,V:INTEGER;
            IDI:IDENTIFIER;
            IDID:DOUBLE_ID;
       (*   WHENEVER THE TERM: PRIOR(XXX) APPEARS   *)
       (*   IN THE VALIDATION TEST,REPLACE IT BY    *)
       (*   THE BACKUP VARIABLE FOR XXX.            *)
       (*   I.E. PRIOR(XXX) WILL BE DELETED AND     *)
       (*   AT THE SAME PLACE  BV000000XXX WILL BE  *)
       (*   INSERTED.............................. *)
       BEGIN
```

```
KK:=0;  Q:=1;
FOR M:=1 TO K-5 DO
BEGIN
  FOR L:=1 TO 5 DO P[L]:=TEST[M+L-1];
  IF P='PRIOR' THEN
  BEGIN
    FOR L:=Q TO M-1 DO
    BEGIN  KK:=KK+1; T[KK]:=TEST[L];  END;
    Q:=M;
    WHILE (TEST[Q]#')') DO Q:=Q+1;   Q:=Q+1;
    N:=Q-1;
    WHILE (TEST[N]#'(') DO N:=N-1;   N:=N+1;
    FOR S:=1 TO ID_LENGTH DO IDI[S]:=' '; S:=1;
    WHILE ( N<Q-1 ) DO
    BEGIN  IDI[S]:=TEST[N]; N:=N+1; S:=S+1;  END;
    FOR U:=1 TO NO_NLOC DO
    BEGIN
      S:=NLVAR_LIST[U];
      IF NL_VAR_TABLE[S].INDICATOR='ID' THEN
      BEGIN
        S:=NL_VAR_TABLE[S].PTR_TO_VDTAB;
        IF VAR_DCL_TABLE[S].VNAME=IDI THEN
        BEGIN
          FOR V:=1 TO ID_LENGTH*2 DO IDID[V]:=' ';
          FOR V:=ID_LENGTH+1 TO ID_LENGTH*2 DO
          IDID[V]:=IDI[V-ID_LENGTH];
          IDI:='BV000000'; BV_NAME(IDI,U);
          FOR V:=1 TO ID_LENGTH DO IDID[V]:=IDI[V];
          GOTO 173;
        END;
      END;
    END;
    173::
    FOR V:=1 TO ID_LENGTH*2 DO
    BEGIN
      IF IDID[V]#' ' THEN
      BEGIN  KK:=KK+1;  T[KK]:=IDID[V];  END;
    END;
  END;
END;
FOR L:=Q TO K DO
BEGIN  KK:=KK+1;  T[KK]:=TEST[L];  END;
FOR L:=1 TO KK DO TEST[L]:=T[L];
END;

(*   MAIN BODY OF VALIDATION FUNCTION    *)
BEGIN
  (*  GENERATE STAEMENTS FOR THE VALIDATION  *)
  (*  FUNCTION HEADING AND ITS BODY.         *)
  TEST:='FUNCTION          :BOOLEAN;  (* VALIDATION *) ';
  FOR I:=1 TO LINE_LENGTH DO BUF[I]:=' ';


  NAME_CREATION(FNT);
  FOR I:=1 TO ID_LENGTH DO TEST[I+9]:=FNT[I];
  FOR I:=1 TO 47 DO BUF[COL+I-1]:=TEST[I];
  PUT_LINE(TEXT2,BUF,PTR2);
  FOR I:=1 TO LINE_LENGTH DO BUF[I]:=' ';
  PRO:='BEGIN    ';
  FOR I:=1 TO 9 DO BUF[COL+I+2]:=PRO[I];
  PUT_LINE(TEXT2,BUF,PTR2);
  TEST:='NOT(FAULTFLAG) AND (                          ';
  FOR I:=1 TO LINE_LENGTH DO BUF[I]:=' ';
  FOR I:=1 TO ID_LENGTH DO BUF[COL+I+3]:=FNT[I];
  BUF[COL+I+3]:=':';  BUF[COL+I+4]:='=';
  I:=COL+I+5;
  FOR J:=1 TO 20 DO BUF[I+J-1]:=TEST[J];
  I:=I+J;  J:=1;
  WHILE (BUFER[J]=' ') DO J:=J+1;
  WHILE (BUFER[J]#' ') DO J:=J+1;
  WHILE (BUFER[J]=' ') DO J:=J+1;
  FOR K:=1 TO 47 DO TEST[K]:=' ';
  K:=1;
  REPEAT
    TEST[K]:=BUFER[J]; K:=K+1; J:=J+1;
    BLANK:=TRUE;
    FOR M:=J TO LINE_LENGTH DO
    BEGIN
      IF BUFER[M]#' ' THEN BLANK:=FALSE;
    END;
  UNTIL (BLANK);
  SUBSTITUTE_PRIOR(TEST,K,KK);
  FOR J:=1 TO KK DO
  BEGIN BUF[I]:=TEST[J]; I:=I+1; END;
  BUF[I]:=')';  BUF[I+1]:=';';
  PUT_LINE(TEXT2,BUF,PTR2);
  COL:=COL-4;
  FOR I:=1 TO LINE_LENGTH DO BUF[I]:=' ';
  BUF[COL]:='E'; BUF[COL+1]:='N'; BUF[COL+2]:='D';
  BUF[COL+3]:=';';
  PUT_LINE(TEXT2,BUF,PTR2);
  PROC_RH_TREE[NODEPTR].PTR_TO_END_DCL:=PTR2-1;
  COL:=COL-3;
  J:=COL;
  FOR I:=1 TO LINE_LENGTH DO BUF[I]:=' ';
  PRO:='$LINKAGE ';
  FOR I:=1 TO 9 DO BUF[I]:=PRO[I];
  IF PROC_RH_TREE[NODEPTR].FIRST_SON#0 THEN
  BEGIN  PUT_LINE(TEXT2,BUF,PTR2);
  END;
  COL:=J;
  PRO:='BEGIN    ';
  FOR I:=1 TO LINE_LENGTH DO BUF[I]:=' ';
  FOR I:=1 TO 9 DO BUF[COL+I-1]:=PRO[I];
```

```
      PUT_LINE(TEXT2,BUF,PTR2);
      TST:='FAULTFLAG:=FALSE;                    ; ';
      FOR I:=1 TO ID_LENGTH DO TST[19+I]:=PVS[I];
      FOR I:=1 TO LINE_LENGTH DO BUF[I]:=' ';
      FOR I:=1 TO 50 DO BUF[COL+I+5]:=TST[I];
      PUT_LINE(TEXT2,BUF,PTR2);
   END;
PROCEDURE OBJBLK_TRANSLATION;
   VAR ENS_CNT:INTEGER;
       PREV_WORD,NEXT_WORD:STRING10;

   PROCEDURE CREATE_END;
      (*   GENERATE THE END STAEMENT FOR THE   *)
      (*   TRANSLATED RECOVERY BLOCK.          *)
      (*   EXAMPLE:                            *)
      (*             1357:END;                 *)
      (*   LABLES ARE 1357,1367,1377,1387,...  *)
      (*   USED FOR RECOVERY BLOCKS AS THEY    *)
      (*   APPEAR IN TEXT....................  *)
      VAR J,K:INTEGER;
      BEGIN
            K:=PROC_RB_TREE[NODEPTR].LABLE;
            FOR J:=I TO LINE_LENGTH DO BUFER[J]:=' ';
            J:=COL;   FILL_ADR_FIELD(BUFER,J,K);
            BUFER[COL+4]:=':';   BUFER[COL+5]:='E';
            BUFER[COL+6]:='N';   BUFER[COL+7]:='D';
            BUFER[COL+8]:=';';
            PUT_LINE(TEXT2,BUFER,PTR2);
      END;

   PROCEDURE GIVE_NEXT_WORD;
      VAR I,J:INTEGER;

      FUNCTION MORE(VAR TX:INTEGER):BOOLEAN;
         VAR J:INTEGER;
         BEGIN
               MORE:=TRUE;
               FOR J:=TX TO LINE_LENGTH DO
               BEGIN
                   IF TEMP_STMT[J] #' ' THEN MORE:=FALSE;
               END;
         END;
      PROCEDURE PASS_QUOTES;
         VAR I,J:INTEGER;
         BEGIN I:=0;
            IF TEMP_STMT[TXT_PTR+1]='''' THEN I:=TXT_PTR+1
            ELSE IF TEMP_STMT[TXT_PTR+2]='''' THEN
            I:=TXT_PTR+2;
            IF I#0 THEN
            BEGIN
              I:=I+1;   J:=1;



              WHILE (J MOD 2 #0) DO
              BEGIN
                 IF TEMP_STMT[I]='''' THEN
                 BEGIN
                    WHILE (TEMP_STMT[I]='''') DO
                    BEGIN  J:=J+1;  I:=I+1;
                    END;
                 END
                 ELSE I:=I+1;
              END;
              TXT_PTR:=I;
              PASS_QUOTES;
           END;
         END;

      PROCEDURE PASS_COMMENTS;
         VAR I:INTEGER;
         BEGIN  I:=TXT_PTR+1;
          IF NOT MORE(I) THEN
          BEGIN
                WHILE(TEMP_STMT[I]=' ') DO  I:=I+1;
                IF((TEMP_STMT[I]='(') AND
                   (TEMP_STMT[I+1]='*')) THEN
                BEGIN I:=I+1;
                      REPEAT  I:=I+1;
                      UNTIL ((TEMP_STMT[I]='*') AND
                             (TEMP_STMT[I+1]=')'));
                      TXT_PTR:=I;
                END;
          END;
        END;
      (*  MAIN BODY OF GIVE NEXT WORD   *)
      (*                                *)
      BEGIN   INDIC:=2;
         PREV_WORD:=NEXT_WORD;
         PASS_QUOTES;
         PASS_COMMENTS;
         IF  MORE(TXT_PTR)  THEN
         BEGIN
             PUT_LINE(TEXT2,TEMP_STMT,PTR2);
             GET_LINE(TEXT1,TEMP_STMT,PTR1);
         END;
         TOKEN:='                    ';  I:=TXT_PTR;
         WHILE((I<LINE_LENGTH) AND
             (NOT(TEMP_STMT[I] IN SYMBOLS))) DO I:=I+1;
         IF I=LINE_LENGTH THEN
         BEGIN PUT_LINE(TEXT2,TEMP_STMT,PTR2);
               GET_LINE(TEXT1,TEMP_STMT,PTR1);
               I:=I;
               WHILE(NOT(TEMP_STMT[I] IN SYMBOLS)) DO I:=I+1;
         END;
        J:=1;
        WHILE(((I<=LINE_LENGTH) AND
              (TEMP_STMT[I] IN SYMBOLS)) DO
        BEGIN TOKEN[J]:=TEMP_STMT[I]; I:=I+1; J:=J+1; END;
        TXT_PTR:=I;
        FOR I:=1 TO 10 DO NEXT_WORD[I]:=TOKEN[I];
        IF NEXT_WORD[1] IN ['0'..'9'] THEN GIVE_NEXT_WORD;
        IF (TOKEN='ELSE_ERROR            ') THEN CEC:=PTR1-4;
     END;

  PROCEDURE CREATE_IF_THEN;
     (*  GENERATE STAEMENTS FOR THE BODY OF  *)
     (*  PROCEDURE CORRESPONDING TO          *)
     (*  THE TRANSLATED RECOVERY BLOCK.      *)
     (*  EXAMPLE:                            *)
     (*      IF VT000001 THEN GOTO 1357      *)
     (*      ELSE BEGIN RS000001;            *)
     (*                 FAULTFLAG:=FALSE;    *)
     (*           END;                       *)
     VAR TEMP1:STRING45;
         TEMP2:STRING47;
         I,J,K:INTEGER;
     BEGIN
       TEMP1:='IF             THEN GOTO         (* EXIT *)      ';
        FOR I:=1 TO ID_LENGTH DO TEMP1[I+3]:=FNT[I];
        FOR I:=1 TO LINE_LENGTH DO BUFER[I]:=' ';
        IF COL+44>LINE_LENGTH THEN COL:=LINE_LENGTH-45;
        FOR I:=1 TO 45 DO BUFER[COL+I-1]:=TEMP1[I];
        K:=PROC_RB_TREE[NODEPTR].LABLE;   J:=COL+22;
        FILL_ADR_FIELD(BUFER,J,K);
        PUT_LINE(TEXT2,BUFER,PTR2);
     TEMP2:='ELSE BEGIN           ; FAULTFLAG:=FALSE; END;     ';
        FOR I:=1 TO ID_LENGTH DO TEMP2[I+11]:=PRV[I];
        FOR I:=1 TO LINE_LENGTH DO BUFER[I]:=' ';
        IF COL+48>LINE_LENGTH THEN COL:=LINE_LENGTH-49;
        FOR I:=1 TO 47 DO BUFER[COL+I]:=TEMP2[I];
        PUT_LINE(TEXT2,BUFER,PTR2);
     END;

  PROCEDURE  ERROR_MESSAGE;
     (*  GENERATE ERROR MESSAGE FOR THE  *)
     (*  CASE THAT ALL ALTERNATES OF THE *)
     (*  RECOVERY BLOCK FAILED.          *)
     VAR TEMP:STRING45;
         TEMP2:STRING47;
         NAME:IDENTIFIER;
         I,J,K:INTEGER;
     BEGIN
       TEMP:='IF             THEN GOTO         (* EXIT *)      ';
        FOR I:=1 TO ID_LENGTH DO TEMP[I+3]:=FNT[I];
        FOR I:=1 TO LINE_LENGTH DO BUFER[I]:=' ';


        IF COL+44>LINE_LENGTH THEN COL:=LINE_LENGTH-45;
        FOR I:=1 TO 45 DO BUFER[COL+I-1]:=TEMP[I];
        K:=PROC_RB_TREE[NODEPTR].LABLE;   J:=COL+22;
        FILL_ADR_FIELD(BUFER,J,K);
        PUT_LINE(TEXT2,BUFER,PTR2);
       TEMP:='ELSE BEGIN                                       ';
        FOR I:=1 TO 45 DO BUFER[COL+I-1]:=TEMP[I];
        PUT_LINE(TEXT2,BUFER,PTR2);
        FOR I:=1 TO LINE_LENGTH DO BUFER[I]:=' ';
TEMP2:='WRITELN(''RECOVERY PROCEDURE               FAILED'');';
        NAME:=PROC_RB_TREE[NODEPTR].PR_RB_NAME;
        FOR J:=1 TO ID_LENGTH DO TEMP2[J+29]:=NAME[J];
        IF COL+57>LINE_LENGTH THEN COL:=LINE_LENGTH-58;
        FOR I:=1 TO 47 DO BUFER[COL+I+10]:=TEMP2[I];
        PUT_LINE(TEXT2,BUFER,PTR2);
        FOR J:=1 TO LINE_LENGTH DO BUFER[J]:=' ';
       TEMP:='FAULTFLAG:=TRUE;                                 ';
       FOR I:=1 TO ID_LENGTH DO TEMP[I+17]:=PRV[I];
       TEMP[I+17]:=';';
        FOR J:=1 TO ID_LENGTH+27 DO BUFER[COL+J-1]:=TEMP[J];
        PUT_LINE(TEXT2,BUFER,PTR2);
        FOR J:=1 TO LINE_LENGTH DO BUFER[J]:=' ';
        BUFER[COL-6]:='E';  BUFER[COL-5]:='N';
        BUFER[COL-4]:='D';  BUFER[COL-3]:=';';
        PUT_LINE(TEXT2,BUFER,PTR2);
        COL:=COL-12;
     END;

  PROCEDURE  TRAVEL;
     (*  WHEN THERE ARE NESTED RECOVERY BLOCKS  *)
     (*  JUST TRANSLATE THE OUTER ONE AND SKIP  *)
     (*  OVER THE INNER ONES.....................*)
     VAR I,J:INTEGER;
     BEGIN
        INDIC:=2;
        WHILE(ENS_CNT>0) DO
        BEGIN
           GIVE_NEXT_WORD;
           IF NEXT_WORD='ENSURE     ' THEN
           BEGIN
              NEED_MORE_PASS:=TRUE;
              FOR J:=1 TO ID_LENGTH DO ID[J]:=PREV_WORD[J];
              I:=HASH(ID);    J:=HASH_PTRS[I];
              WHILE(ID # HASH_TABLE[J].PR_NAME) DO
              BEGIN
                 REPEAT  I:=(I+1) MOD HASH_PTRS_LENGTH;
                 UNTIL   (HASH_PTRS[I] #0);
                 J:=HASH_PTRS[I];
              END;
              OLDPTR:=HASH_TABLE[J].PTR_TO_PROC_TREE;
              PROC_RB_TREE[OLDPTR].ENTRY:=PTR2-2;
           END;
           IF NFXT_WORD='BEGIN      ' THEN ENS_CNT:=ENS_CNT+1
           ELSE IF NEXT_WORD='END       '
           THEN ENS_CNT:=ENS_CNT-1;
        END;
     END;
  (*   MAIN BODY OF OBJBLK_TRANSLATION    *)
  BEGIN
      OLDPTR::=NODEPTR;    INDIC:=2;
      GIVE_NEXT_WORD;
      WHILE(NEXT_WORD # 'ELSE_ERROR') DO
      BEGIN
         IF NEXT_WORD='ELSE_BY    ' THEN
         BEGIN
            CREATE_IF_THEN;
            GET_LINE(TEXT1,TEMP_STMT,PTR1);
            GET_LINE(TEXT1,TEMP_STMT,PTR1);
            GIVE_NEXT_WORD;
            IF NEXT_WORD='BEGIN      ' THEN
            BEGIN
                  ENS_CNT:=1;
                  TRAVEL;
                  PUT_LINE(TEXT2,TEMP_STMT,PTR2);
            END
            ELSE PUT_LINE(TEXT2,TEMP_STMT,PTR2);
            GET_LINE(TEXT1,TEMP_STMT,PTR1);
            GIVE_NEXT_WORD;
            IF NEXT_WORD='$LINKAGE   ' THEN
               GET_LINE(TEXT1,TEMP_STMT,PTR1)
            ELSE GOTO 250;
         END
         ELSE IF NEXT_WORD='BEGIN      ' THEN
         BEGIN
            ENS_CNT:=1;
            TRAVEL;
         END;
         GIVE_NEXT_WORD;
         250:;
      END;
      ERROR_MESSAGE;
      CREATE_END;
   END;


(*****************************************)
(*                                       *)
(*          MAIN BODY OF PASS_2          *)
(*                                       *)
(*****************************************)

BEGIN


   INITIATE_PASS2;
   WHILE(NEED_MERGE) DO
   BEGIN
      (*   WHEN NEED_MERGE IS TRUE, WE HAVE   *)
      (*   NEW TYPE DECLARATIONS GENERATED    *)
      (*   BY PASS 1. THESE NEW TYPE DCL'S    *)
      (*   ARE IN A FILE CALLED TEXT2. MERGE  *)
      (*   THEM INTO THE MAIN TEXT.......... *)
      N:=POS_NEW_TYPE+1;
      FCOPY(TEXT1,TEXT3,BUFER,PTR1,PTR3,N);
      IF DCL_TYPE THEN
      BEGIN  FOR I:=1 TO LINE_LENGTH DO BUFER[I]:=' ';
             BUFER[2]:='T'; BUFER[3]:='Y';
             BUFER[4]:='P'; BUFER[5]:='E';
             PUT_LINE(TEXT3,BUFER,PTR3);
      END;
      N:=1;
      WHILE(BUFER[N] =' ') DO N:=N+1;
      WHILE(NOT EOF(TEXT2)) DO
      BEGIN
         FOR I:=1 TO N-1 DO BUF[I]:=' '; J:=N;
         WHILE((NOT EOLN(TEXT2))AND (J<=LINE_LENGTH)) DO
         BEGIN  READ(TEXT2,BUF[J]); J:=J+1; END;
         READLN(TEXT2);
         FOR I:=J TO LINE_LENGTH DO BUF[I]:=' ';
         PUT_LINE(TEXT3,BUF,PTR3);
      END;
      NEED_MERGE:=FALSE;
   END;
   PTR2:=1;  REWRITE(TEXT2);
   WHILE  (NODE#0)  DO
   BEGIN
      IF FIRST_CALL THEN DCL_FAULTFLAG;
      FIRST_CALL:=FALSE;
      NEW_NODE(SEARCH_DIRECTION);

      (*   NODE=0 ---> TREE HAS BEEN VISITED  *)
      (*   NODE=1 ---> PROCEDURE              *)
      (*   NODE=2 ---> TERMINAL NODE IN TREE  *)
      (*   NODE=3 ---> REC.BLOCK NODE         *)
      (*   BY NODE WE MEAN A NODE ON THE      *)
      (*   PROC/RB TREE...................... *)
      CASE  NODE  OF
      1:    BEGIN
               N:=PROC_RB_TREE[NODEPTR].ENTRY;
               FCOPY(TEXT1,TEXT3,BUFER,PTR1,PTR3,N);
               SEARCH_DIRECTION:=1;
               PROC_RB_TREE[NODEPTR].MARK:='V';
            END;

      2:    BEGIN
```

```
               I:=PROC_RB_TREE[NODEPTR].YOUNGER_BROTHER;
               WHILE (I=0) DO
               BEGIN NODEPTR:=PROC_RB_TREE[NODEPTR].FATHER;
                     IF NODEPTR=1 THEN
                     BEGIN  NODE:=0;  GOTO 550;  END
                     ELSE
                     I:=PROC_RB_TREE[NODEPTR].YOUNGER_BROTHER;
               END;
               J:=PROC_RB_TREE[NODEPTR].YOUNGER_BROTHER;
               IF PROC_RB_TREE[J].MARK='V'
                  THEN BEGIN  NODEPTR:=I; SEARCH_DIRECTION:=1;
                       END
                  ELSE SEARCH_DIRECTION:=2;
               550::
            END;

      3:    BEGIN
               FIND_ELDER_BROTHER;
               CASE  ELDER_BROTHER  OF
               1: BEGIN
                     (*  ELDER BROTHER IS NULL  *)
                     (*  OR A PROCEDURE.......  *)
                     K:=PROC_RB_TREE[NODEPTR].FATHER;
                     N:=PROC_RB_TREE[K].PTR_TO_END_DCL+1;
                     FCOPY(TEXT1,TEXT3,BUFER,PTR1,PTR3,N);
                     INDIC:=3;
                     GET_LINE(TEXT1,BUFER,PTR1);
                     J:=I0;
                     FILL_ADR_FIELD(BUFER,J,PTR2);
                     PUT_LINE(TEXT3,BUFER,PTR3);
                     PROC_RB_TREE[K].PTR_TO_END_DCL:=PTR3;
                     N:=PROC_RB_TREE[NODEPTR].ENTRY;
                     FCOPY(TEXT1,TEXT3,BUFER,PTR1,PTR3,N);
                     INDIC:=3;
                     GET_LINE(TEXT1,BUFER,PTR1);
                     RB_TRANSLATION(NODEPTR);
                     CREATE_LINKAGE;
                     NODEPTR:=OLDPTR;
                     IF NOT YOUNGER(NODEPTR) THEN
                     PUT_LINE(TEXT2,LASTLINE,PTR2);
                     SEARCH_DIRECTION:=1;
                  END;

               2:  BEGIN
                     (*  ELDER BROTHER IS REC.BLOCK  *)
                     N:=PROC_RB_TREE[NODEPTR].ENTRY;
                     FCOPY(TEXT1,TEXT3,BUFER,PTR1,PTR3,N);
                     INDIC:=3;
                     GET_LINE(TEXT1,BUFER,PTR1);
                     J:=I0;   I:=10;
                     WHILE(LASTLINE[I] #' ') DO I:=I+1;


                     I:=I-1;
                     CONVERT_DECIMAL(LASTLINE,J,I,K);
                     I:=PROC_RB_TREE[NODEPTR].FATHER;
                     PROC_RB_TREE[I].PTR_TO_END_DCL:=K;
                     RB_TRANSLATION(NODEPTR);
                     CREATE_LINKAGE;
                     NODEPTR:=OLDPTR;
                     SEARCH_DIRECTION:=1;
                  END
                END;
             END
        END;
   END;
   (*  CHANGE THE NODE INDICATOR OF   *)
   (*  THOSE REC.BLOCK NODES,WHICH    *)
   (*  ARE TRANSLATED INTO PROCEDURE  *)
   FOR I:=1 TO LIM-1 DO
   BEGIN   N:=RB_TO_PROC[I];
          PROC_RB_TREE[N].INDICATOR:='PR';
   END;
   INDIC:=1;
   REPEAT   GET_LINE(TEXT1,BUFER,PTR1);
            PUT_LINE(TEXT3,BUFER,PTR3);
   UNTIL   (EOF(TEXT1));
END;

PROCEDURE PRINT_PROC_TREE;
   VAR I:INTEGER;
   BEGIN
      WRITELN(' .... PROCEDURE TREE,VARYING COMPONENTS ....');
      WRITELN;WRITELN;
      WRITELN('PROC NAME',' INDICATOR','  END OF DCL',
              '  ENTRY','  MARK');   WRITELN;
      FOR I:=1 TO NEXT_PROC_NODE-1 DO
      WRITELN(PROC_RB_TREE[I].PR_RB_NAME,'      ',
              PROC_RB_TREE[I].INDICATOR,
              PROC_RB_TREE[I].PTR_TO_END_DCL:11,
              PROC_RB_TREE[I].ENTRY:3,'      ',
              PROC_RB_TREE[I].MARK);
      WRITELN;WRITELN;
   END;
PROCEDURE  MERGE_INTXT(VAR TEXT3,TEXT2,TEXT1:TEXT_FILE);
   (*  MERGE TEXT3 INTO TEXT2 --->TEXT1  *)
   VAR I,T,K:INTEGER;
   BEGIN
     INDIC:=1;
     RESET(TEXT2);   RESET(TEXT3);   REWRITE(TEXT1);
      PTR2:=1;          PTR3:=1;          PTR1:=1;
     REPEAT
        READ_LINE(TEXT2,BUFER,PTR2);
        IF BUFER[1]='$' THEN
```

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

```
              BEGIN
                K:=1;
                IF ((BUFER[10]=' ') AND (BUFER[11]=' ') AND
                    (BUFER[12]=' ') AND (BUFER[13]=' '))
                   THEN K:=0;
                T:=1;
                FOR I:=1 TO LINE_LENGTH DO
                BEGIN  IF TEMP_STMT[I]=':' THEN
                   BEGIN  IF((TEMP_STMT[I+1]='E') AND
                             (TEMP_STMT[I+2]='N') AND
                             (TEMP_STMT[I+3]='D')) THEN T:=0;
                   END;
                END;
                IF ((K#0) AND (T#0)) THEN
                BEGIN
                  WHILE (NOT EOF(TEXT3)) DO
                  BEGIN  READ_LINE(TEXT3,BUFER,PTR3);
                         IF BUFER[1]='$' THEN GOTO 650
                            ELSE SAVE_LINE(TEXT1,BUFER,PTR1);
                  END;
                END
                ELSE IF((K#0) AND (T=0)) THEN
                     SAVE_LINE(TEXT1,BUFER,PTR1);
              END
              ELSE BEGIN
                        SAVE_LINE(TEXT1,BUFER,PTR1);
                        TEMP_STMT:=BUFER;
                   END;
            650:;
           UNTIL  (EOF(TEXT2));
         END;

(**************************************************)
(*                                                *)
(*     MAIN BODY OF MAIN PROGRAM.............     *)
(*                                                *)
(**************************************************)

BEGIN
   PASS_1;
   IF  NO_NEW_TYPES>0 THEN NEED_MERGE:=TRUE  ELSE NEED_MERGE:=FALSE;
   FIRST_CALL:=TRUE;
   PRINT_PROC_TREE;
   PASS_2(TEXT1,TEXT2,TEXT3);
   PRINT_PROC_TREE;
   IF NOT(NEED_MORE_PASS) THEN
   BEGIN  MERGE_INTXT(TEXT2,TEXT3,TEXT1);  GOTO 2000;  END;
   PASS_2(TEXT2,INTX4,TEXT1);
   PRINT_PROC_TREE;
   IF NOT(NEED_MORE_PASS) THEN


   BEGIN  MERGE_INTXT(INTX4,TEXT1,TEXT2);
          MERGE_INTXT(TEXT2,TEXT3,TEXT1);
          GOTO 2000;
   END;
   PASS_2(INTX4,INTX5,TEXT2);
   PRINT_PROC_TREE;
   IF NOT(NEED_MORE_PASS) THEN
   BEGIN
          MERGE_INTXT(INTX5,TEXT2,INTX4);
          MERGE_INTXT(INTX4,TEXT1,TEXT2);
          MERGE_INTXT(TEXT2,TEXT3,TEXT1);
          GOTO 2000;
   END;
   SW:=1;
   WHILE(NEED_MORE_PASS) DO
   BEGIN
      MERGE_INTXT(TEXT2,TEXT1,TEXT4);
      IF SW=1 THEN
      BEGIN  PASS_2(INTX5,INTX4,TEXT2);
             SW:=2;
      END
      ELSE  BEGIN  PASS_2(INTX4,INTX5,TEXT2);
                   SW:=1;
            END;
   END;
   IF SW=1 THEN  MERGE_INTXT(INTX5,TEXT2,TEXT1)
          ELSE   MERGE_INTXT(INTX4,TEXT2,TEXT1);
   MERGE_INTXT(TEXT1,TEXT4,TEXT2);
   MERGE_INTXT(TEXT2,TEXT3,TEXT1);
   2000:;
END.
```

Appendix B. Sample test-runs of FT-PASCAL preprocessor

B.1 An FT-PASCAL program

```
TYPE   P=RECORD
             X:INTEGER;
             Y:REAL
          END;
       A=ARRAY[1..5] OF P;
VAR I,J:INTEGER;
       B:ARRAY[1..10] OF P;
BEGIN
   I:=1;
   J:=5;
   ENSURE     I=J
   BY         I:=I+1;
   ELSE_BY    I:=I+4;
   ELSE_BY    B[2].X:=I+5;
   ELSE_ERROR;
   WRITELN(I)
END.
```

B.2 Translation of the program in B.1 by FT-PASCAL preprocessor

```
TYPE  P=RECORD
           X:INTEGER;
           Y:REAL
         END;
      A=ARRAY[1..5] OF P;
      TP000001=RECORD
                       X          :INTEGER ;
                       Y          :REAL
                 END;
      TP000002= ARRAY[1..10] OF P                                ;
VAR I,J:INTEGER;
      B:ARRAY[1..10] OF P;
      FAULTFLAG:BOOLEAN;
   PROCEDURE RB000001;
      VAR
          BV000001I           : INTEGER ;
          BV000002B           : TP000002;
      PROCEDURE  SA000001;        (* SAVE *)
         BEGIN
             BV000001I          :=I           ;
             BV000002B          :=B;
         END;
      PROCEDURE  RS000001;        (* RESTORE *)
         BEGIN
             I          :=BV000001I          ;
             B:=BV000002B          ;
         END;
      FUNCTION VT000001 :BOOLEAN;     (* VALIDATION *)
         BEGIN
             VT000001:=NOT(FAULTFLAG) AND ( I=J );
         END;
      BEGIN
            FAULTFLAG:=FALSE;  SA000001 ;
            I:=I+1;
            IF VT000001 THEN GOTO 1357     (* EXIT *)
             ELSE BEGIN RS000001; FAULTFLAG:=FALSE; END;
            I:=I+4;
            IF VT000001 THEN GOTO 1357     (* EXIT *)
             ELSE BEGIN RS000001; FAULTFLAG:=FALSE; END;
            B[2].X:=I+5;
            IF VT000001 THEN GOTO 1357     (* EXIT *)
            ELSE BEGIN
                         WRITELN('RECOVERY PROCEDURE  RB000001 FAILED');
                         FAULTFLAG:=TRUE; RS000001;
                  END;
     1357:END;
BEGIN
  I:=1;
  J:=5;
    RB000001;
  WRITELN(I)
END.
```

B.3 An FT-PASCAL program

```
TYPE  P=RECORD
          X:INTEGER;
          Y:ARRAY[1..3] OF CHAR;
          Z:REAL
        END;
VAR A:ARRAY[1..10] OF INTEGER;
   N,I:INTEGER;
     REC:P;
FUNCTION  ALL_SORT:BOOLEAN;
  VAR I:INTEGER;
  BEGIN
    ALL_SORT:=TRUE;
    FOR I:=2 TO N DO
    BEGIN
      IF A[I] < A[I-1] THEN
      BEGIN
        ALL_SORT:=FALSE;
      END;
    END;
  END;

PROCEDURE  SORT1;
  VAR I,J,K:INTEGER;
  BEGIN
  WRITELN('ENTERED SORT 1');
  WRITELN('A WAS AS FOLLOWS:');
  FOR I:=1 TO N DO WRITELN(A[I]);
   REC.X:=2;
  WRITELN;  WRITELN;
     FOR I:=1 TO N-1 DO
     BEGIN
        FOR J:=I+1 TO N DO
        BEGIN
           IF A[I] < A[J] THEN
           BEGIN
              K:=A[I];
              A[I]:=A[J];
              A[J]:=K;
           END;
        END;
     END;
     WRITELN('RETURNED FROM SORT 1');
     WRITELN('A IS NOW AS FOLLOWS:');
     FOR I:=1 TO N DO WRITELN(A[I]);
     WRITELN;  WRITELN;
  END;

PROCEDURE  SORT2;
  VAR I,J,K:INTEGER;
  BEGIN
  WRITELN('ENTERED SORT 2');
  WRITELN('A WAS AS FOLLOWS:');
  FOR I:=1 TO N DO WRITELN(A[I]);
  REC.Z:=3;
  WRITELN;  WRITELN;
     FOR I:=1 TO N-1 DO
     BEGIN
        FOR J:=I+1 TO N DO
        BEGIN
           IF A[I] > A[J] THEN
           BEGIN
              K:=A[I];
              A[I]:=A[J];
              A[J]:=K;
           END;
        END;
     END;
```

- continued -

```
        WRITELN('RETURNED FROM SORT 2');
        WRITELN('A IS NOW AS FOLLOWS:');
        FOR I:=1 TO N DO WRITELN(A[I]);
        WRITELN;  WRITELN;
     END;

   BEGIN
     N:=10;
     WRITELN('INITIAL ELEMENTS OF A');
     FOR I:=1 TO N DO  READLN(A[I]);
     FOR I:=1 TO N DO  WRITELN(A[I]);
     WRITELN;  WRITELN;

     ENSURE    ALL_SORT
     BY        SORT1;
     ELSE_BY   SORT2;
     ELSE_ERROR;

     WRITELN('FINAL ELMENTS OF A');
     FOR I:=1 TO N DO  WRITELN(A[I]);
   END.
```

B.4 Translation of the program in B.3 by FT-PASCAL preprocessor

```
   TYPE  P=RECORD
             X:INTEGER;
             Y:ARRAY[1..3] OF CHAR;
             Z:REAL
           END;
           TP000002= ARRAY[1..3] OF CHAR                             ;
           TP000001=RECORD
                              X           :INTEGER ;
                              Y           :TP000002;
                              Z           :REAL
                         END;
           TP000003= ARRAY[1..10] OF INTEGER                         ;
   VAR A:ARRAY[1..10] OF INTEGER;
      N,I:INTEGER;
        REC:P;
        FAULTFLAG:BOOLEAN;
   FUNCTION  ALL_SORT:BOOLEAN;
     VAR I:INTEGER;
     BEGIN
       ALL_SORT:=TRUE;
       FOR I:=2 TO N DO
       BEGIN
         IF A[I] < A[I-1] THEN
         BEGIN
           ALL_SORT:=FALSE;
         END;
       END;
     END;

   PROCEDURE  SORT1;
     VAR I,J,K:INTEGER;
     BEGIN
     WRITELN('ENTERED SORT 1');
     WRITELN('A WAS AS FOLLOWS:');
     FOR I:=1 TO N DO WRITELN(A[I]);
      REC.X:=2;
     WRITELN;  WRITELN;
        FOR I:=1 TO N-1 DO
        BEGIN
           FOR J:=I+1 TO N DO
           BEGIN
              IF A[I] < A[J] THEN
              BEGIN
                 K:=A[I];
                 A[I]:=A[J];
                 A[J]:=K;
              END;
           END;
        END;
        WRITELN('RETURNED FROM SORT 1');
        WRITELN('A IS NOW AS FOLLOWS:');
        FOR I:=1 TO N DO WRITELN(A[I]);
        WRITELN;  WRITELN;
     END;
```

- continued -

```
PROCEDURE  SORT2;
  VAR I,J,K:INTEGER;
  BEGIN
  WRITELN('ENTERED SORT 2');
  WRITELN('A WAS AS FOLLOWS:');
  FOR I:=1 TO N DO WRITELN(A[I]);
  REC.Z:=3;
  WRITELN;  WRITELN;
     FOR I:=1 TO N-1 DO
     BEGIN
        FOR J:=I+1 TO N DO
        BEGIN
           IF A[I] > A[J] THEN
           BEGIN
              K:=A[I];
              A[I]:=A[J];
              A[J]:=K;
           END;
        END;
     END;
     WRITELN('RETURNED FROM SORT 2');
     WRITELN('A IS NOW AS FOLLOWS:');
     FOR I:=1 TO N DO WRITELN(A[I]);
     WRITELN;  WRITELN;
  END;

   PROCEDURE RB000001;
      VAR
          BV000001REC      : INTEGER ;
          BV000002A        : TP000003;
          BV000003REC      : REAL     ;
     PROCEDURE  SA000001;       (* SAVE *)
        BEGIN
             BV000001REC       :=REC.X;
             BV000002A         :=A          ;
             BV000003REC       :=REC.Z;
        END;
     PROCEDURE  RS000001;       (* RESTORE *)
        BEGIN
            REC.X:=BV000001REC        ;
            A        :=BV000002A      ;
            REC.Z:=BV000003REC        ;
        END;
     FUNCTION VT000001 :BOOLEAN;     (* VALIDATION *)
        BEGIN
            VT000001:=NOT(FAULTFLAG) AND ( ALL_SORT );
        END;
     BEGIN
           FAULTFLAG:=FALSE;   SA000001 ;
           SORT1;
           IF VT000001 THEN GOTO 1357     (* EXIT *)
            ELSE BEGIN RS000001; FAULTFLAG:=FALSE; END;
           SORT2;
           IF VT000001 THEN GOTO 1357     (* EXIT *)
           ELSE BEGIN
                      WRITELN('RECOVERY PROCEDURE  RB000001 FAILED');
                      FAULTFLAG:=TRUE; RS000001;
                END;
     1357:END;
BEGIN
  N:=10;
  WRITELN('INITIAL ELEMENTS OF A');
  FOR I:=1 TO N DO  READLN(A[I]);
  FOR I:=1 TO N DO  WRITELN(A[I]);
  WRITELN;  WRITELN;

     RB000001;

  WRITELN('FINAL ELMENTS OF A');
  FOR I:=1 TO N DO  WRITELN(A[I]);
END.
```

B.5 Input data for the program in B.3

```
00100    9
00200    1
00300    8
00400    2
00500    7
00600    3
00700    6
00800    4
C0900    5
01000    9
```

B.6 Result of the execution of the program in B.4 with the data in B.5

After SORT1 fails, SORT2 succeeds in producing acceptable result.

```
INITIAL ELEMENTS OF A              ENTERED SORT 2
          9                        A WAS AS FOLLOWS:
          1                                  9
          8                                  1
          2                                  8
          7                                  2
          3                                  7
          6                                  3
          4                                  6
          5                                  4
          9                                  5
                                             9

ENTERED SORT 1                     RETURNED FROM SORT 2
A WAS AS FOLLOWS:                  A IS NOW AS FOLLOWS:
          9                                  1
          1                                  2
          8                                  3
          2                                  4
          7                                  5
          3                                  6
          6                                  7
          4                                  8
          5                                  9
          9                                  9

RETURNED FROM SORT 1
A IS NOW AS FOLLOWS:               FINAL ELMENTS OF A
          9                                  1
          9                                  2
          8                                  3
          7                                  4
          6                                  5
          5                                  6
          4                                  7
          3                                  8
          2                                  9
          1                                  9
```

Part III

PROGRAMMER-TRANSPARENT COORDINATION OF RECOVERING PARALLEL PROCESSES

by

K. H. Kim

PROGRAMMER-TRANSPARENT COORDINATION OF RECOVERING PARALLEL PROCESSES

by K. H. Kim

ABSTRACT: A new approach to coordination of cooperating parallel processes, each capable of error detection, rollback, and retry, is presented. Error detection, rollback, and retry in a proccess are specified by a well-structured language construct called recovery block. Coordination of processes is needed to prevent a disastrous avalanche of process rollbacks. In contrast to the previously studied approaches that require the program designer to coordinate the recovery block structures of interacting processes, the new approach relieves the program designer of that burden. It instead relies upon an intelligent processor system (that runs processes) capable of establishing and discarding recovery points of interacting processes in a well coordinated manner such that (1) a process never makes two consecutive rollbacks without making a retry between the two, and (2) every process rollback becomes a minimum-distance rollback. Following the discussion of the underlying philosophy of the new approach, basic rules of reducing storage and time overhead in such a processor system are discussed. For specific illustration, systems in which processes communicate through monitors, are considered and then an intelligent processor system equipped with new monitor mechanizations termed fault-tolerant monitors and new process stores termed extended recovery caches is developed.

Index Terms: rollback propagation, process interaction, programmer-transparent coordination, recovery block, fault-tolerant monitor, extended recovery cache.

1. Introduction

As a pragmatic supplement to other widely practiced approaches to obtaining reliable real-time software, incorporation of run-time error detection and recovery capabilities into large-scale software is becoming an increasingly common practice [H1,K1,M1,R1,R2,W1,Y1,Y2]. To successfully exploit the potential of this approach, it is essential that program redundancy (specifying error detection and recovery functions) be embedded within the program in a well-structured form so that the clarity of the overall program is not degraded [D1,H3,P1,R1]. A language construct called recovery block (RB) or fault-tolerant block was introduced by Horning et al to support structured incorporation of redundancy into a program [H3]. It has the following syntactic structure: ensure V by O_1 else-by O_2 else-by --- else-by O_n else-error, where V denotes the validation test, O_1 the primary object block, and O_k $(2 \leq k \leq n)$ the alternate object blocks. Readers familiar with the semantics of the RB and with the efficient storage scheme called recovery cache [A1,H3] supporting execution of RBs, may skip the next four paragraphs.

All the object blocks in an RB specify computations aimed at producing the same or approximately the same result. A process executes the validation test V on exit from an object block to confirm that the result of the object block execution is acceptable. If it is acceptable, the process exits from the RB. If it is not, the process enters the next alternate object block. Also, the process enters the next alternate object block if the underlying processor system detects an error (e.g., divide-by-zero) while the process is inside an object block.

Before an alternate object block is entered, the process state is restored to the state that existed just before entry to the primary object block. That is, the process rolls back to the recovery point

(RP) established on entry to the RB [A1,C1,H3]. Each variable that was assigned a new value by the rejected execution is restored to its original value. The underlying processor system automatically performs this "assignment reversal". To enable this, the first assignment to a non-local variable v during execution of an object block is preceded by the recording of the original value of v, denoted by PRIOR(v), in the recovery cache. Actually PRIOR(v) may also be needed by the validation test of the RB. These (first) assignment records need to be kept until the RB is successfully exited. When an RB is exited, the RP established on entry to the RB may be discarded.

For example, consider a program in Figure 1a. Figure 1b shows a snapshot of the recovery cache taken when primary object block $O_{2.1}$ is in execution. As shown, there is a stack, called a cache stack, used for saving the original values. Similar to the main stack, the cache stack is also divided into regions, one region for each nested RB in the "active" state (i.e., an RB that has been entered but not exited). The top region of the cache stack in Figure 1b contains names (representing logical addresses) and previous values of the non-local variables that have been modified during execution of the current object block $O_{2.1}$ (i.e., Y2,X1,X2). Similarly, the bottom region of the cache stack contains the previous value of non-local variable X1 which had been modified by execution of object $O_{1.1}$ before $O_{2.1}$ was entered. Figure 1b also shows a flag field in the main stack. The flag indicates whether the original value of the associated variable has already been saved since the current object block was entered. Thus the flags attached to Y2, X1, X2 in the main stack are currently set.

If the result produced by execution of $O_{2.1}$ is rejected by validation test V_2, then the top region C_2 of the cache stack can be used to reset the main stack to the state that existed on entry to

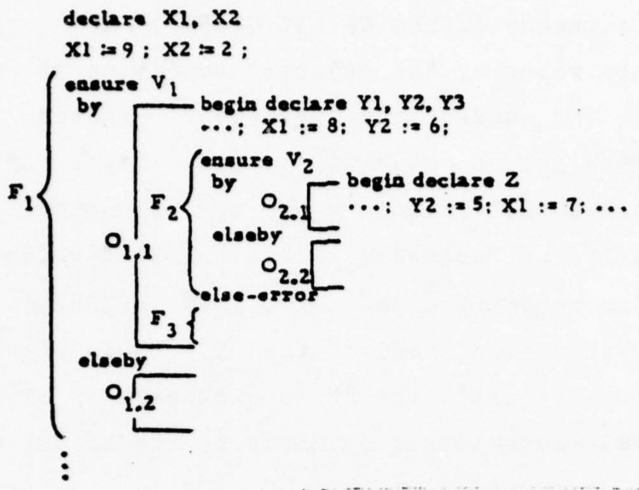


Figure 1a A block-structured fault-tolerant program

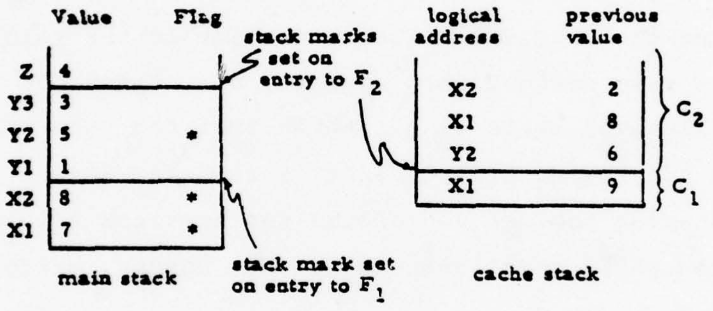


Figure 1b Recovery cache during execution of 1a

RB F_2. If it passes the test, execution of F_2 is complete and C_2 is merged into C_1 such that the result will contain previous values of those variables that are non-local to $O_{1.1}$ and have been modified since $O_{1.1}$ was entered. Thus the result will be a single region containing (X1,9) and (X2,2). Flags in the main stack are also adjusted such that only the flags of X1 and X2 are set. Therefore, creation of a region in the cache stack can be viewed as the establishment of a recovery point (RP) and the number of regions in the cache stack at any time is equal to the number of RPs existing at that time.

In a system of cooperating parallel processes each of which is independently structured by RBs, process rollback and retry becomes greatly complicated by the possibility that rollback of a process X may cause the rollback of other processes which have interacted with X[R1,R3,S1]. A process Y often has to roll back into an RB from which it already exited successfully, because other processes which interacted with Y while Y was inside the RB, fail their validation tests later (i.e., after Y exited the RB). This means that a process cannot always discard an RP on successful exit from an RB. In addition, a disastrous avalanche of rollback propagations between processes, called a domino effect [R1], could occur. To control the domino effect, either the program designer must carefully coordinate RPs (by coordinating the RB structures) [R1,R3] or the processor system that runs processes must be designed to automatically manage RPs properly and coordinate the rollbacks of interacting processes.

Randell proposed a language construct called conversation to aid the programmer in coordinating the RB structures, thereby supporting a programmed (RP) coordination approach [R1]. In this paper a programmer-transparent coordination approach relying upon an intelligent processor system is explored, and some principles useful for its practical realization are discussed. To provide a specific example, an intelligent processor system that supports processes communicating through monitors [B1,D2,H2] is developed. The processor system uses new monitor mechanizations termed fault-tolerant monitors and new process stores termed extended recovery caches. (The fault-tolerant monitor is not directly related to the recoverable monitor that was developed in [S1] for a different purpose.) Again, the basic principles related to coordination of the RP establishments and rollbacks of processes are valid independently of the mechanism by which cooperating processes communicate. A short review of the basic characteristics of a monitor is given in the next paragraph.

A monitor consists of a shared data structure and all the operations, called monitor procedures, that processes can perform on it. No more than one monitor procedure of the same monitor may be in execution at any instant. A process X which has gained exclusive possession of a monitor and entered one of the monitor procedures, can release the monitor in two ways: one is to exit from the monitor procedure and the other is to execute a "wait(q)" where q is the name of a queue, called a condition queue, into which process X will enter to sleep. Without loss of generality, the capacity of each condition queue is assumed to be one. A process Y which enters a monitor procedure while another process X is sleeping in condition queue q may waken the process X by executing a "signal(q)". A process can execute this signal operation only as it exits a monitor procedure. A process checks, at various stages during execution of a monitor procedure, to see if the current state of the monitor satisfies the prerequisite condition for the next operation; if the condition is not satisfied, the process will enter a condition queue.

In section 2, basic problems arising in communication among parallel fault-tolerant processes are delineated and then the underlying philosophies of the programmer-transparent coordination approach are discussed. Sections 3 and 4 develop a fault-tolerant monitor and section 5 discusses a possible extension. Proofs of two lemmas stated in section 3 are given in Appendix A.

2. A programmer-transparent approach to coordination of recovering parallel processes

For convenience in exposition and specific illustration of basic principles, the systems of processes dealt with in the rest of this paper are assumed to have the following characteristics. For ease of reference, the assumptions, the notations, and the definitions that are developed in this paper are numbered A1, A2, etc, N1, N2, etc, and D1, D2, etc, respectively.

Assumption:

(A1) All the processes are created during system initialization and, once created, either exist forever or terminate together.

(A2) Processes can communicate only through monitors, and every resource shared among processes must be contained within a monitor. (This assumption is removed in section 5.)

(A3) There must be a procedure (known to the underlying processor system) by which the state of a monitor can be recorded at any instant and a procedure by which a recorded state can be restored later.

(A4) Monitor procedures do not contain wait or signal instructions. (This assumption is removed in section 3.5.)

Assumption A2 implies that the only operations that can be performed on a shared resource are monitor procedures. This restrictive assumption is adopted mainly to simplify the presentation of the basic ideas of the proposed scheme and it is removed in section 5. Assumption A3 is trivially met if the contents of shared variables (including condition queues) in a monitor completely represent the state of the monitor, because at any time the contents can be readily copied into another memory region and later reloaded, if necessary. Recording and restoring a state of a special resource (e.g., disk) contained in a monitor may require special procedures unique to the resource, perhaps supplied by the program designer. This then would be the only burden put on the program designer.

Figure 2a depicts a history of a system of two fault-tolerant processes communicating through a single monitor. The figure uses the following notations.

Notation:

(N1) The processes progress downward.

(N2) A bracket represents an RB execution (RBE) and each short wavy line represents a recovery point (RP) of a process. The bottom end of a bracket represents an execution of the associated validation test.

(N3) A shaded column represents the history of the state of a monitor, and a change in the direction of shading represents a state change.

(N4) A horizontal line represents execution of a monitor procedure. A line directed toward a shaded column represents a monitor update operation (i.e., execution of a monitor procedure which only changes the state of a monitor without examining the state at all), while a line directed toward a process from a shaded column represents a monitor reference operation (i.e., execution of a monitor procedure which does not change but examines the state of a monitor). In most cases a monitor procedure contains both reference and update operations. Execution of such a procedure is treated as a monitor reference operation immediately followed by a monitor update operation.

Process A in Figure 2a produces information at A.3 for other processes and stores into monitor M. A little later process B picks up this information (or a part of it) at B.4. Process B passes its validation test at B.5. If process A fails at A.p, then it will roll back to RP A.1 and revoke the information it sent out between A.1 and A.p. (As part of this rollback, process A must restore monitor M to the state that existed prior to A.3.) Revocation of A.3 should cause process B to roll back to an RP established prior to B.4, even though

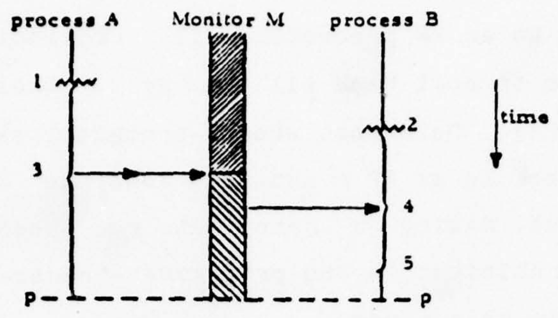


Figure 2a A system history

B has already passed its own validation test. A programmed validation test cannot detect all possible errors. Therefore, if process A fails, process rollback is propagated to process B due to the revocation of the information sent out by A. This type of rollback propagation is called an R-propagation in this paper.

On the other hand, if process B in Figure 2a fails at B.5 and rolls back to RP B.2, then the question arises as to whether process A should roll back. The information received at B.4 may have caused the failure of B. Therefore, process B may be suspicious of the integrity of process A and ask A to roll back to an RP prior to A.p. This type of rollback propagation is due to the suspicion by process B and called an S-propagation.

When S-propagations are permitted, an avalanche of rollback propagations may occur unless the program designer takes great care in coordinating the RB structures of interacting processes. For example, if process A in Figure 2b fails at A.p, then process A may be suspicious of the integrity of the information received at A.22. Process B may have supplied the information either in full at B.21 or in parts at B.5, B.13, and B.21. Some part of the information may even have been deposited in the monitor by process A itself at A.1, A.9 or A.17. Upon receiving a request from process A to roll back to an RP preceding B.21, process B may in turn be suspicious of the integrity of the information received at B.18 and thus request process A to roll back to an RP preceding A.17. Continuing this way, both processes will have to roll back all the way to their beginnings (a domino effect occurs). Note that when S-propagations are permitted, a process can roll back to an RP r and then continue to roll back to another RP without making a retry at r. Because of this, S-propagations are prohibited in the programmer-transparent coordination approach explored in this paper.

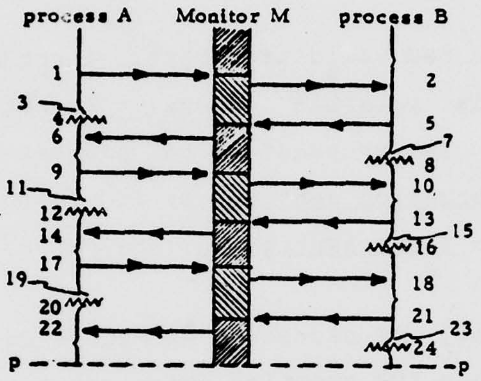


Figure 2b A system history

Allowing only R-propagations means that a process which sends out incorrect information is solely responsible for detection and correction of this error. Under this stategy a process can discard the RP established on initiation of an RB on passing the associated validation test if the process did not receive information from other processes but only sent out information during execution of the RB. For example, process A in Figure 2a which does not know the speed of the progress of process B can discard RP A.1 on passing validation test A.p. In a sense, process A gives a posteriori accreditation of the information sent out between A.1 and A.p on passing A.p and then no other processes can challenge the integrity of the information.

When only R-propagations are permitted, it is possible to prevent a process from making two consecutive rollbacks (without a retry after the first rollback), by establishing RPs immediately before certain, but not all, monitor reference operations. These RPs are in addition to the RPs established on initiation of RB executions (RBEs). The RPs established immediately before monitor references are called branch RPs, while the RPs established on initiation of RBEs are called base RPs. Establishment of branch RPs is transparent to the program designer. For example, process B in Figure 2c established a branch RP immediately before monitor reference B.7. Such an RP will be referred to by the name (e.g., B.7) of the immediately following monitor operation in the rest of this paper. Process A also established branch RP A.3. If process A fails validation test A.p and revokes monitor update A.6, then process B rolls back to RP B.7 and no more rollback propagation will ensue. In this case, process B makes a minimum-distance rollback. If branch RPs B.7 and A.3 had not been established, process B would have rolled back to base RP B.1 and revoked monitor update B.2, which in turn would have caused process A to roll back to an RP preceding A.3; process A would thus have made two consecutive rollbacks, first to RP A.5 and then to the RP preceding A.3, without a retry from A.5.

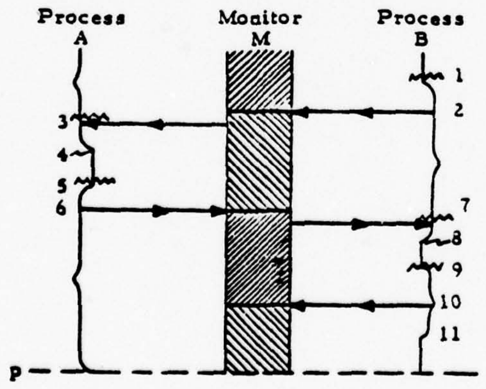


Figure 2c Establishment of two branch RPs A.3 and B.7

Figure 2c also reveals an additional aspect of monitor update accredidation. Before process A executes validation test A.p, process B has passed validation test B.11 and thus has given a posteriori accreditation of monitor update B.10. Note that this accreditation is indirectly voided when process A fails at A.p and causes process B to roll back to B.7, although the accreditation issued is not directly challenged.

In short, the essence of the programmer-transparent coordination approach is to prohibit S-propagations and to establish certain branch RPs in addition to base RPs, thereby ensuring that a process never makes two consecutive rollbacks. A process need also save the states of a monitor immediately before certain, but not all, monitor update operations in order to be able to restore the monitor if the updates are revoked later. The approach is practical only if at all times the number of RPs and the number of previous states of monitors that each process needs to maintain is manageable. All the management functions, including maintenance of RPs, restoration of monitors, and coordination of process rollbacks, must be automatically handled by the underlying processor system. Basic principles that can be gainfully exploited in practical implementation of the programmer-transparent coordination approach are discussed in the next section.

3. Fault-tolerant monitor and extended recovery caches in systems using one monitor

It was mentioned in the preceding section that the feasibility of the programmer-transparent coordination approach is largely dependent upon the reduction of the time and storage overhead required by an intelligent processor system. Two major sources of overhead are in creation and discard of (1) RPs and (2) records of previous states of monitors. Basic principles useful for reduction of this overhead are now discussed. To simplify the presentation, this section deals only with the systems using one monitor. Figure 3 depicts such a system. Furthermore, most illustrations are made with simple two-process systems.

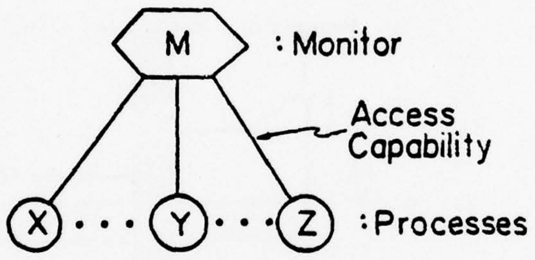


Figure 3 A system of processes communicating through one monitor

3.1 Minimal recovery point (RP) establishment

To get an intuitive idea, consider Figure 4a. Process B established a branch RP at the beginning of monitor reference B.4 since the information existing then in the monitor was subject to possible future revocation (due to the possible failure of the RBE initiated at A.1). However, it was not necessary for process B to establish additional RPs at monitor references between B.6 and B.j. This is because any information picked up from M between B.4 and B.j is revoked only when process A rolls back to A.1, and thus the references (made between B.6 and B.j) are voided together with reference B.4. Therefore, an on-going RBE of a process X causes another process Y to create at most one branch RP, and process Y must maintain the RP at least until the RBE (of X) deceases. If process A in Figure 4a passes validation test A.p, then RP B.4 may be discarded. (Communication of validation results and other notices among processes is discussed in section 3.3.1.) Base RP B.2 can not be discarded even after validation test B.k succeeds as long as branch RP B.4 remains. This is because once process B rolls back to branch RP B.4, it can fail at B.k in which case it needs to roll back to base RP B.2. In a sense, the RBE that was initiated at B.2 is not completely validated even after B.k until the RBE that was initiated at A.1 is completely validated at A.p and branch RP B.4 is discarded.

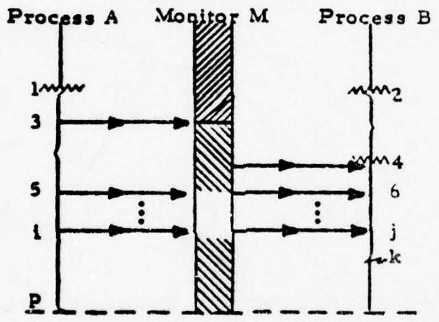


Figure 4a A system history

So that the conditions for establishment and discard of RPs can be stated precisely, the following notations and terms are introduced.

Notation: (N5) An RBE is represented by [a:] or [:b], where a and b are the starting and the ending execution points, respectively, of the RBE. For example, [B.2:] and [:B.k] in Figure 4a represent the same RBE.

Difinition:

(D1) When a monitor M is updated by a process X during an RBE [X.a:], the RBE [X.a:] becomes a potential recaller of monitor M and holds that right until [X.a:] deceases. For example, [A.1:] in Figure 4a is a potential recaller of M from A.3 to A.p. Monitor M is said to be rollback-chained or R-chained to the base RP established at the starting point X.a of RBE [X.a:].

(D2) When a monitor M has no potential recallers, it is said to be rollback-free.

(D3) If a process Y references a monitor M which is R-chained to base RP X.a, then process Y becomes R-chained to RP X.a through monitor M, and thus RBE [X.a:] becomes a potential recaller of process Y. If process Y becomes R-chained to RP X.a during an RBE [Y.c:], then RBE [Y.c:] is said to be R-chained to RP X.a. Thus RBE [X.a:] is a potential recaller of RBE [Y.c:].

(D4) An RBE of a process X is a potential recaller of process X itself.

In Figure 4b RBE [A.2:] in becomes a potential recaller of M at its first update (A.3) of M, and RBE [B.1:] of process B becomes R-chained to base RP A.2 at the first reference (B.4) to M after M became R-chained to the RP A.2. Monitor M has two potential recallers [A.2:] and [B.1:] immediately after B.5. By definition D4, RBE [B.1:] is a potential recaller of process B between B.1 and B.p.

Definition:

(D5) The set of all the potential recallers of a process X (or a monitor M) at a given time t is called the potential recaller set (PRS) of X (or M) at t, and is denoted by PRS(X,t) (or PRS(M,t)). The second argument t may be omitted if there is no ambiguity. Similarly, the PRS of an RBE [X.a:] at t is defined and is denoted by PRS([X.a:],t), where t may be omitted if there is no ambiguity.

(D6) The potential recaller closure (PR*) of an RBE [X.a:] at a

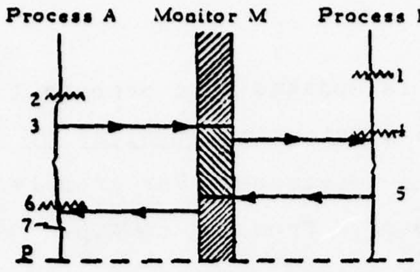


Figure 4b A system history

given instant, denoted by PR*([X.a:]), is a set of RBEs defined as follows: every potential recaller of RBE [X.a:] is a member of PR*([X.a:]); if an RBE e is a member of PR*([X.a:]), then every potential recaller of e is also a member of PR*([X.a:]).

(D7) An RBE [Z.e:] which is not a potential recaller of RBE [X.a:] but a member of PR*([X.a:]), is called an <u>indirect potential recaller</u> of RBE [X.a:].

(D8) The set of all the indirect potential recallers of an RBE [X.a:] is called the <u>indirect potential recaller set (IPRS)</u> of RBE [X.a:] and is denoted by IPRS([X.a:]).

The PRS of an RBE [X.a:] can be a proper subset of PR*([X.a:]) because PR*([X.a:]) may increase after process X passes the validation test of RBE [X.a:] (and thus PRS([X.a:]) is fixed). For example, process C in Figure 4c already passed the validation test of RBE [C.1:] at C.6. Thus PRS([C.1:]) was fixed at C.6 and includes on-going RBE [B.2:] of process B. When RBE [B.2:] became R-chained to RP A.5 of another process A later at B.9, RBE [A.5:] became a member of PR*([C.1:]) but it is not a potential recaller of [C.1:].

<u>Definition</u>:

(D9) An RBE [:X.b] is said to be <u>partially validated</u> if validation test X.b has been successful but there remain potential recallers of [:X.b] (i.e., there are RBEs of other processes which became potential

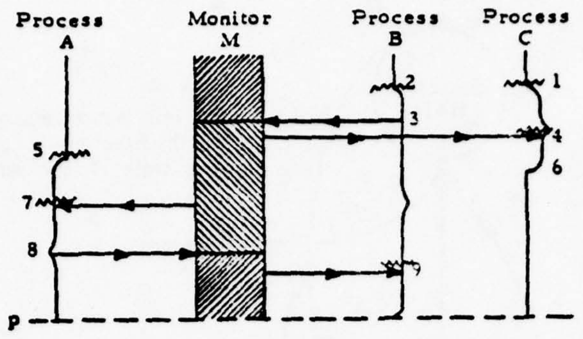


Figure 4c An indirect potential recaller [A.5:] of RBE [C.1:]

recallers of process X during [:X.b] and have not deceased).

(D10) An RBE [:X.b] is <u>completely validated</u> if (1) validation test X.b has been successful and (2) either PR*([:X.b]) is empty or every member of PR*([:X.b]) has been completed with passing of the associated validation test.

An informal definition of "complete validation" is validation of every aspect of an RBE. Therefore, once an RBE [X.a:] is completely validated, there will never be any need for process X to roll back to RP X.a. In the case of Figure 4b, complete validation of RBE [A.2:] or [B.1:] consists of validation tests A.7 and B.p. RBE [A.2:] is first partially validated at A.7. Then RBE [A.2:] becomes completely validated together with [B.1:] at B.p when every member of PR*([A.2:]) = PR*([B.1:]) = {[A.2:],[B.1:]} has obtained the partially validated status. On the other hand, RBE [C.1:] in Figure 4c is not completely validated even when validation test B.p succeeds, because PR*([C.1:]) contains on-going RBE [A.5:]. Figure 5 depicts various states that a newly initiated RBE may go through.

The rule for maintaining the PRSs of processes and of the monitor stems from definitions D1 - D5, and is stated below.

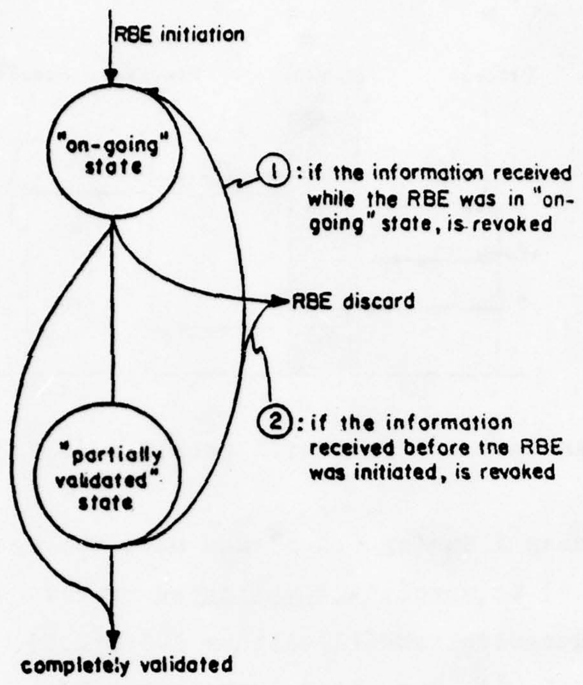

RBE initiation
"on-going" state
1 : if the information received while the RBE was in "on-going" state, is revoked
RBE discard
2 : if the information received before the RBE was initiated, is revoked
"partially validated" state
completely validated

Figure 5 Life cycle of an RBE

Rule: (R1) "PRS management"

(R1.1) RBE initiation: when a process X is about to initiate an RBE at X.a, PRS(X) is updated to PRS(X) U {[X.a:]} where U is a set-union operator.

(R1.2) Monitor update: when a process X is about to update a monitor M, PRS(M) is updated to PRS(M) U PRS(X).

(R1.3) Monitor reference: when a process X is about to reference a monitor M, PRS(X) is updated to PRS(X) U PRS(M).

(R1.4) Complete validation of an RBE: If an RBE [X.a:] has been completely validated, [X.a:] is removed from the PRSs of processes and from the PRS of the monitor.

(R1.5) Discard of an RBE: If an RBE [:X.b] is discarded due to a failure at or before X.b, [:X.b] is removed from the PRSs of processes and from the PRS of the monitor.

By using the PRSs of processes and the monitor, a rule for correctly establishing RPs can be stated as follows.

Rule: (R2) "RP establishment and discard": when the PRS of process X, PRS(X), is expanded by incorporating a set E of new members, a new RP is established by X. E is called the PRS of the RP. The RP may be discarded only when all the members of its PRS E have been removed from PRS(X).

Note that the PRSs of RPs are mutually disjoint and that the PRS of a process X is the same as the union of the PRSs of all the RPs maintained by X at that time. In addition, the PRS of RBE [:X.b] at X.b is equal to the union of the PRSs of the RPs that were established during [:X.b] and have remained. As an illustration of the above rule, PRS(A) in Figure 4a is expanded at A.1 and thus a base RP is established at A.1. Similarly, PRS(B) is expanded at B.4 and thus a branch RP is established at B.4. When validation test A.p succeeds, RBE [A.1:] is completely validated and thus is removed from PRS(A), PRS(B), and PRS(M). Since the PRSs of both RPs A.1 and B.4 are now empty, the two RPs are discarded. RBE [B.2:] has also been completely validated and RP B.2 is discarded. In the case of Figure 2c, RP B.9 is discarded when validation test B.11 succeeds but RPs B.7, B.1, A.5, and A.3 are all discarded when validation test A.p succeeds.

The above PRS management rule R1 is not complete yet. See Figure 6a. Under the present rule process B establishes an RP at B.4. Therefore, if process A fails at A.p and rolls back to A.1, then process B rolls back to B.4 and resumes execution. However, the information produced at B.3 has been lost and thus the information process B acquires at B.4 after resuming execution may be incorrect. A logical solution to this problem is to establish an RP at (the beginning of) B.3 instead of B.4. The problem then is how to recognize the need for RP establishment at a monitor update such as B.3. Note that PRS(M) contains an RBE which is not contained in PRS(B) at B.3. Therefore a solution is to modify the rule R1 as follows.

Modified rule: (R1.2) Monitor update: when a process X is about to update a montor M, both PRS(X) and PRS(M) are updated to PRS(X) U PRS(M).

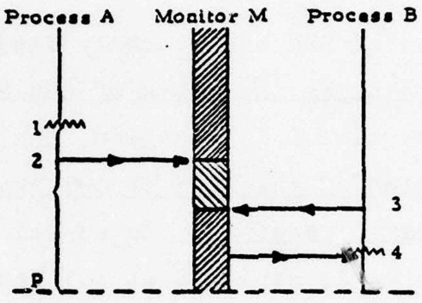


Figure 6a Incorrect establishment of a branch RP

In the case of Figure 6a, PRS(B) is a null set before B.3 but it becomes {[A.1:]} at the beginning of B.3. Therefore, by the RP management rule R2, an RP is established at the beginning of B.3. Since PRS(B) does not change at B.4, no RP is established there.

The following statement can be made about this RP management rule.

Lemma: (L1) Under rule R2 processes maintain the minimum number of RPs required to enable every rollback to be made to the most recent valid execution point, provided RBEs can be removed from the PRSs as soon as they are completely validated or discarded. (Proof in Appendix A)

An example of an RP having more than one potential recaller is RP B.7 in Figure 6b which has two potential recallers, [A.1:] and [A.5:], until A.10. This means that RP B.7 must remain intact until A.11 (even after [A.5:] is successfully completed at A.10). Therefore, the number of RPs maintained by a process X is always less than or equal to the cardinality of PRS(X). PRS(X) is of course a subset of on-going or partially validated RBEs of the processes in the system.

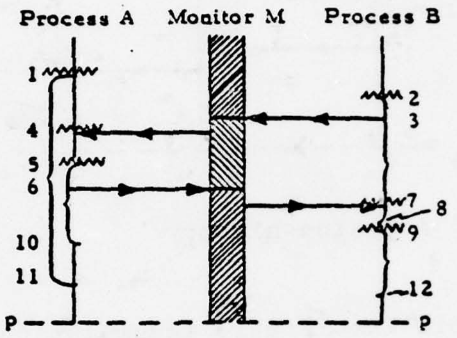


Figure 6b A branch RP (B.7) having two potential recallers

3.2 Reduction of the number of RPs

Although no redundant RPs are maintained under the RP management rule R2 given in the preceding section, the number of RPs maintained at one time could, in many cases, exceed the tolerable limit (imposed mainly by storage space available). For example, consider Figure 7a which looks similar to but possesses the opposite characteristics of Figure 2b from the RP management point of view. (Every RBE in Figure 2b is completely validated as its associated validation test succeeds.) Each time a validation test succeeds, it results only in a partial validation because there is a potential recaller which is active at that time. Therefore, no RPs could have been discarded by A.p. If process A passes validation test A.p, then all the RPs (except B.25) can be discarded.

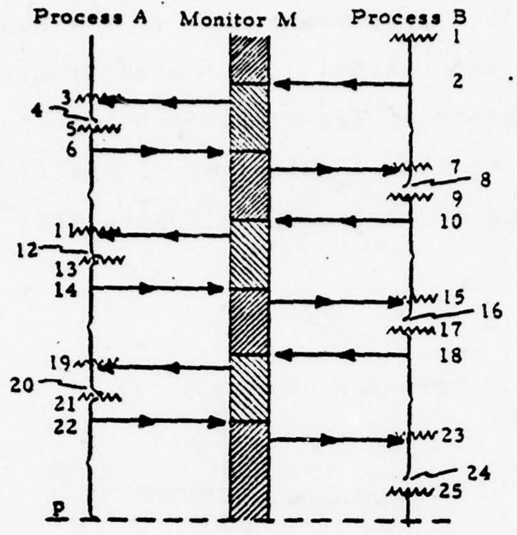


Figure 7a A system history

In order to avoid intolerable accumulation of partially validated RBEs and associated RPs in a process X, process X can safely remove old RPs which have low probability of being used. For example, RP B.1 or B.7 in Figure 7a is considered to have a comparatively low probability of being used because it can be used only when all the subsequent RBEs (by both processes A and B) that have been partially validated were actually wrong. Therefore, process B may remove B.7 and thus sacrifice the capability of rolling back to the most recent valid execution point B.7 in the case where RBE [A.5:] is discarded. Process B must retain base RP B.1 after discarding RP B.7. If RBE [A.5:] is later completely validated, then RP B.1 can also be discarded. If [A.5:] is discarded, then process B can immediately declare that RBE [B.1:] has failed and then roll back to B.1 as if it had made a retry from B.7 and failed the validation test of [B.1:]. This will cause process A to roll back to A.3. Therefore, process B as well as process A can still be recovered as long as it maintains the oldest RP, although it sacrificed the ability to make a minimum-distance rollback every time. In other words, a process can trade increase of rollback distance (in case of less probable rollbacks) for reduction of RPs.

Definition: (D11) A base or branch RP r is said to be supported by a base RP q if r was established during RBE [q:]. For example, RP B.7 in Figure 7a is supported by RP B.1. Also RP A.5 in Figure 6b is supported by RP A.1.

A two-part rule for safe reduction of RPs is now given.

Rule: (R3) "RP reduction"
(R3.1) A branch RP r can be removed if (1) its immediately preceding RP q is a base RP and (2) r is supported by q. Once removed, r is called a defunct branch of q. If process X is required to roll back to defunct branch r, it will immediately notify other processes and the monitor that RBE [q:] has failed and then it will roll back to q.
(R3.2) A base RP r can be removed if (1) its immediately preceding RP q is another base RP and (2) there are no remaining RPs supported by r. Once removed, r and its defunct branches become defunct branches of q. When process X removes r, it notifies other processes and the monitor that the role of RBE [r:] as a potential recaller is taken over by RBE [q:]. If process X is required to roll back to defunct branch r, it will immediately communicate that RBE [q:] has failed and then it will roll back to q.

The immediate effect of removing a branch RP is local to the process. However, the effect of removing a base RP may propagate to other processes. For example, suppose that process B in Figure 7a first removed branch RPs B.7 and B.15 and then removed base RP B.9. Now whenever process B is required to roll back to B.9, it has to roll back to B.1. This means that process A will never roll back to A.11; instead it will be required to roll back to A.3. In other words, the role of [B.9:] as a potential recaller has been taken over by [B.1:].

Therefore, if a base RP X.j has been made a defunct branch of another base RP X.i, RBE [X.j:] must be removed from the PRSs of processes and of the monitor. Removal of [X.j:] from the PRS of a process Y may enable discard of RPs in the process Y. Figure 7b shows the result after process B removes the five RPs B.7, B.15, B.9, B.23 and B.17 in sequence. Again RPs A.3, A.21 and B.1 can be discarded when validation test A.p succeeds.

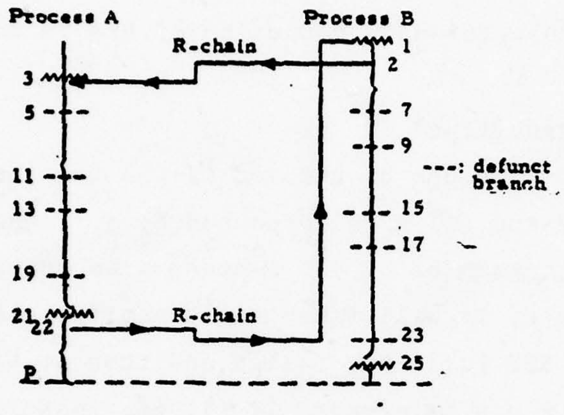


Figure 7b Result after process B removes five RPs from 7a

Implementation of the rules for RP establishment, discard, and reduction is discussed in the next section.

3.3 Implementation of the RP management rule

3.3.1 Store structures, PRS maintenance, and mailbox operations

Figure 8a depicts store structures in an abstract form. Each process or monitor owns an independent store space. A process store (PS) which is an extended recovery cache, consists of a main stack, a cache store, and a table PRST containing the PRS of the process. As in the recovery cache shown in Figure 1b, a main stack provides space for local variables of internal procedures and monitor procedures that have been entered by a process but not exited. A cache store, which is an extension of a cache stack (shown in Figure 1b), contains information necessary for validation and rollback. Although the PRS of a process as a whole is contained in the table PRST, the PRS of each RP is recorded in a cache store (detailed in section 3.3.2). Thus information on the PRS of a process as a whole can also be assembled from the PRS records of RPs. Table PRST within a process store is redundant in a sense but its use is motivated by an efficiency consideration. Updating table PRST always accompanies updating some of the PRS records of RPs kept in a cache store. Note that table PRST and PRS records of RPs are transparent to the program designer; the processor system automatically provides and manages these.

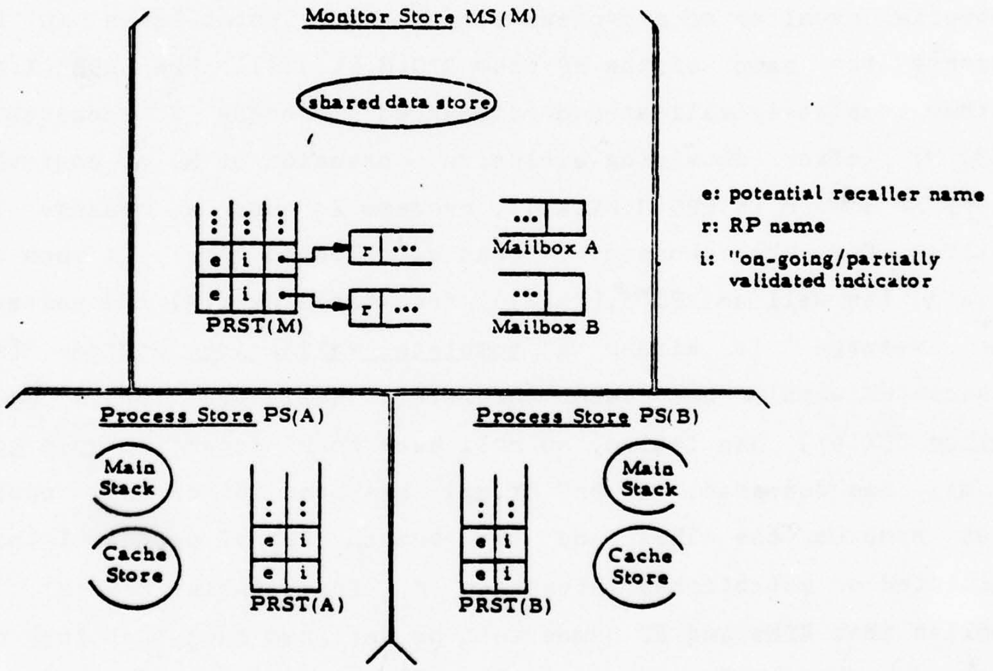


Figure 8a Process store and monitor store

A monitor store (MS) consists of a <u>shared data store</u>, a <u>table PRST</u>, and <u>mailboxes</u> which are queues of messages addressed to processes. Also attached to each entry (i.e., potential recaller name) in table PRST(M) of monitor M is a queue of RP names.

Notation:
(N6) The process store of process X is denoted by PS(X) and the monitor store of monitor M is denoted by MS(M). The mailbox of process X within MS(M) is denoted by MB(X,M), where the second argument M may be omitted if there is no ambiguity.
(N7) A queue of RP names attached to the entry for RBE [X.a:] in table PRST(M) is denoted by RPQ(M,[X.a:]).

Again table PRST and mailboxes in a monitor store are transparent to the program designer. These resources are used as follows.

When a potential recaller [X.a:] of monitor M becomes a new potential recaller of a process Y, process Y establishes an RP and inserts the name of the RP into RPQ(M,[X.a:]). When RBE [X.a:] is either completely validated or discarded, process X accesses table PRST(M) (after obtaining exclusive possession of M, of course). For every RP name r in RPQ(M,[X.a:]), process X puts a message in the mailbox for the process Y that established RP r. It then deletes [X.a:] (as well as RPQ(M,[X.a:])) from table PRST(M) and releases M. The message is either a complete validation notice "[X.a:] associated with r has been completely validated" or a rollback notice "[X.a:] has failed, so roll back to r" depending upon how RBE [X.a:] has deceased. If RBE [X.a:] has been discarded, process X also removes the RBEs and the branch RPs of process Y that were initiated or established later than r, from table PRST(M). (This implies that RBEs and RP names must be assigned such that they provide the ages of the RBEs and RPs.) Then later when process Y gains the monitor for some reason (e.g., update, reference, notification to the monitor of the decease of an RBE, etc), it checks the mailbox first. If there is a message, process Y picks it up, releases the monitor, and interprets the message before regaining the monitor to take the intended action. When process Y regains the monitor, it again checks the mailbox first. Therefore, monitor acquisition and mail check form a single inseparable operation.

Note that there is a variable time gap between the decease of an RBE and the receipt of its notice by other processes because each process checks a mailbox at its own pace rather than being interrupted when mail is deposited. This means that a process could keep a defunct RBE in its table PRST for some time, but this cost seems well justified, considering the complexity of an entirely interrupt-driven implementation.

However, a situation may arise where a process X never accesses a monitor M after a rollback notice is put in the mailbox MB(X,M). Therefore, it is not possible to completely do away with an interrupt. A process interrupt used to inform a process of the existence of messages in a mailbox can be implemented in various ways. In this paper, it is simply assumed that a system contains a programmer-transparent "watchdog" process which periodically examines all the mailboxes and can force any process in the system to check its mailbox if the process has one or more messages that have been in the mailbox for a time exceeding the preset limit.

When a process Z passes the validation test of an RBE [Z.a:], it needs to determine whether RBE [Z.a:] has been partially validated or completely validated. To make this decision, process Z in general needs to check the status of every member of PR*([Z.a:]) (refer to definition D10). Therefore, a process must know not only the PRS but also the PR* of its RBE that consists of both the PRS and the IPRS (refer to definitions D6 and D8). As will be shown in section 3.3.2, the IPRS of each RBE is recorded in the cache store. Whereas a process Z learns of its new potential recallers without any delay and its table PRST(Z) as well as its record of the PRS of each RP is always up-to-date, the process Z learns of new indirect potential recallers of its RBE [Z.a:] with some delay. This is because process Z is informed of new indirect potential recallers through the notices sent by other processes. To be more specific, when RBE [Y.c:], a potential recaller of RBE [Z.a:], has been partially validated, process Y identifies all the members of PR*([Y.c:]) known to itself by that time and attaches the (possibly incomplete) information on PR*([Y.c:]) to a partial validation notice (i.e., a notice of the success of validation test Y.d) sent to process Z. Upon receiving the notice, process Z updates not only its record of the status of [Y.c:] (kept in both table PRST(Z) and its PRS record of an RP) but also its record of IPRS([Z.a:]) kept in its cache store. Since information on PR*([Z.a:]) is not collected at one time, a question may arise as to whether there is danger that process Z will incorrectly judge its RBE to have been completely validated while there is an on-going RBE of another process which is actually a member of PR*([Z.a:]) but not yet known to process Z. The following lemma settles this question with a proof that a process always judges the validation status of its RBEs correctly.

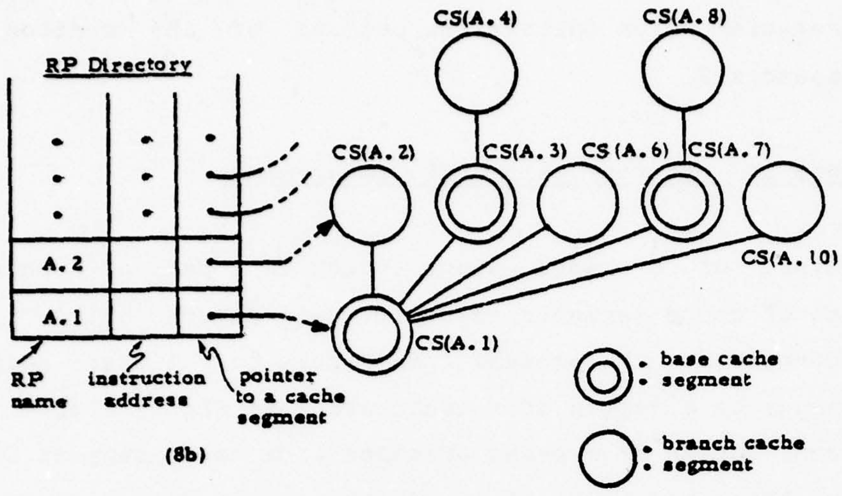
Lemma: (L2) The members of PR*([Z.a:]) known to process Z at a particular time cannot all have the "partially validated" status until all the members of PR*([Z.a:]) become known to process Z with the "partially validated" status. (Proof in Appendix A)

In summary, a process accesses a monitor store not only for an update of or a reference to the shared data store but also for updating table PRST(M) and putting messages in mailboxes. (The latter accesses are transparent to the program designer.) A prerequisite for taking any of these actions is that the process obtain exclusive possession of the monitor and ensure that its mailbox is empty. If the mailbox is not empty, the process must pick up all the messages and immediately release the monitor. A complete procedure (P1) for message interpretation which follows the release of the monitor is described in Appendix B.

3.3.2 Cache store and recovery point (RP) management

The structure of a cache store (which is a part of a process store) is a tree of cache segments as depicted in Figure 8b. Figure 8b actually corresponds to process A in Figure 8c. A cache segment roughly corresponds to a region of a cache stack in Figure 1b. That is, establishment of an RP r means creation of a cache segment CS(r) while discard of RP r means merging of CS(r) into its immediate predecessor cache segment. There exists a <u>one-to-one correspondence between an RP and a cache segment</u>; in implementation, both the RP and the cache segment may be referred to by the same identifier. (Furthermore, an RBE may be represented by the name of the base RP established at its beginning and thus each potential recaller of a process or a monitor may also be referred to by a base RP name in implementation.) The terms "base cache segments" and "branch cache segments" carry meanings analogous to base and branch RPs. There is a table, called <u>RP directory</u>, in which an entry consists of an RP name, the corresponding instruction address, and a pointer to the associated cache segment.

The tree structure represents the <u>support</u> relation among RPs. For example, cache segment CS(A.6) in Figure 8b is linked to CS(A.1) because RP A.6 is supported by RP A.1. This tree structure facilitates safe reduction of RPs by the rule R3 if the number of RPs exceeds a limit determined by storage space available. In addition, since this tree provides information on the relative ages of the existing RPs, it facilitates locating the immediate predecessor of every cache segment and thereby facilitates merging of cache segments. For example, if RP A.6 is to be discarded in Figure 8b, the immediate



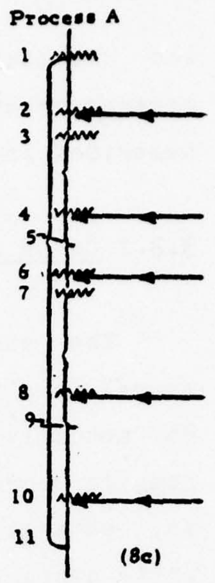


Figure 8b A cache store

Figure 8c A process owning the cache store in 8b

predecessor of CS(A.6) is found to be CS(A.4) from the tree and thus merging of CS(A.6) into CS(A.4) takes place.

More detail on a cache segment is given in Figure 8d. A cache segment CS(r) contains a tree linkage, a base/branch indicator, either PRS(r) if RP r is a branch RP or IPRS([r:]) if RP r is a base RP, and variable assignment records (already shown in Figure 1b). It also contains a "monitor snapshot" that will be described in the next section. Figure 8d also shows that a special record which is a pointer to a region which has been separated out of the main stack, may be contained in the assignment records area. To illustrate such a situation, note that cache segment CS(A.4) is used from A.4 to A.6 (in Figure 8c). A region of the main stack that corresponds to [:A.5] is discarded at A.5. This region must be saved in CS(A.4) because rollback to A.4 includes restoration of the region in the main stack with the values that existed at A.4. This example also points out that local variables of RBE [:A.5] must be treated as non-local variables from A.4 to A.5. Therefore, local variables of RBE [:A.5] can be restored to the values at A.4 by relinking the region saved (at A.5) in CS(A.4) into the main stack and then undoing the assignments recorded (between A.4 and A.5) in CS(A.4).

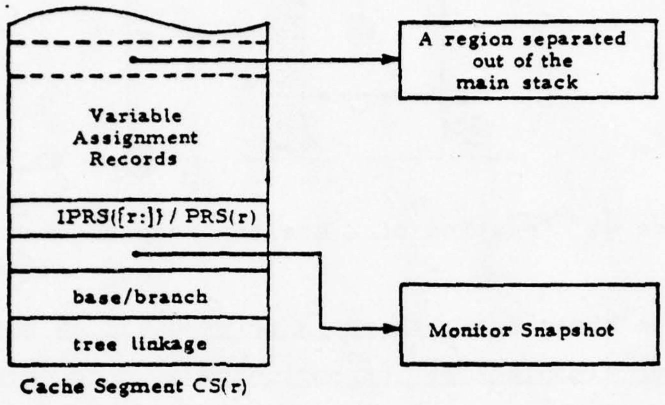


Figure 8d A cache segment

3.4 Minimal number of monitor snapshots

We now turn to the problem of saving and restoring monitor states. See Figure 9a. When process A fails validation test A.p and rolls back to A.1, process B rolls back to B.4. In addition, monitor M must be restored to the state that existed immediately before the first update A.3 by the failed RBE [A.1:]. In order to prepare for this monitor restoration, the responsible process A must record the original state of M before it modifies M at A.3. Unlike the main stack of a process which is recorded incrementally by making assignment records, the recording of a monitor state is performed by creating a snapshot of the entire shared data store and keeping the snapshot in a cache segment of a saving process.

Notation: (N8) A wavy line crossing a shaded column which represents the state history of a monitor M, represents creating a snapshot of M by a process.

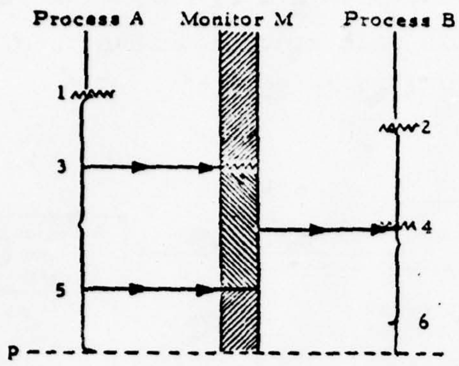


Figure 9a Creation of a monitor snapshot (at A.3)

On the other hand, <u>there is no need for a process to take a snapshot of a monitor at the beginning of a reference to the monitor</u>. To illustrate this, if process B in Figure 9a failed validation test B.6 and rolled back to B.2 to make a retry, process A would not be aware of the rollback of B. Figure 9b shows such a system history. As shown, process B makes a monitor reference (B.4') again during the retry. If monitor operation A.5 were a monitor reference, then the monitor state referenced at B.4' (during the retry) would be identical to the state referenced at B.4 (during the previous unsuccessful try). In such case, creating a snapshot of the monitor at the beginning of monitor reference B.4 was clearly unnecessary. If A.5 is a monitor update as shown, then either the information deposited at A.5 is not needed for reference B.4' (or B.4) or reference B.4 was executed incorrectly. This is because of the asynchronism among processes in a system of cooperating parallel processes. Each process must be designed to wait in a condition queue if the monitor does not contain the desired information. If process B should have waited from A.4 until process A supplied the desired information at A.5, then the monitor will contain all the desired information at reference B.4' during the retry. On the other hand, if the monitor contained all the desired information at B.4, then at monitor update A.5 process A would not know whether process B has already made a reference B.4 and thus would not destroy the information desired at B.4 (unless the process A is faulty). In addition, the information that process A supplies at A.5 will not be needed at B.4'. Thus the monitor will again contain all the desired information at B.4'. Therefore, a monitor reference never accompanies a monitor state recording.

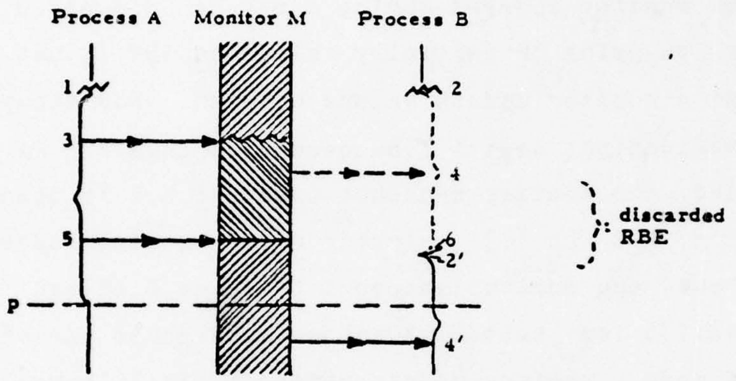


Figure 9b Monitor reference (B.4) not accompanying a monitor recording

Note in Figure 9a that monitor update A.5 is not accompanied by a monitor recording. This is because the most recently established base RP is A.1 and process A has already made a monitor update along with a snapshot before A.5. In other words, any spontaneous rollback of process A occurring after A.5 will be made to base RP A.1 (or to an earlier execution point) and thus will involve restoration of the monitor to the state that existed before the first update A.3 by RBE [A.1:] which precedes A.5. The rule for recording monitor states is stated below.

<u>Rule</u>: (R4) "<u>Monitor recording rule</u>": A snapshot of a monitor must be taken when (and only when) an RBE is about to become a potential recaller of the monitor (at a monitor update). Such an

update is called an R-chaining update.

In systems using one monitor, every R-chaining update is the first update made by a process X during its RBE [X.a:]; in systems using multiple monitors, an update which is not the first to be made by a process during its RBE can be an R-chaining update as will be shown in section 4.

Once an RBE e is completely validated, the monitor snapshots taken at monitor updates during e may be discarded unless there is another on-going or partially validated RBE f that encloses e and had not made a monitor update before e began. For example, see Figure 9c. When validation test A.7 succeeds and thus RBE [A.4:] is completely validated, the monitor snapshot taken at A.6 is discarded because the enclosing RBE [A.1:] already made a monitor update at A.3. On the other hand, the monitor snapshot taken at B.12 can not be discarded when validation test B.13 succeeds, because the enclosing RBE [B.9:] had not made a monitor update before [B.11:] began. That is, merging of cache segment CS(B.11) into CS(B.9) should involve moving the snapshot taken at B.12 into CS(B.9).

3.5 Handling of wait and signal instructions

Assumption A.4 is removed now. A monitor procedure may involve one or more executions of a wait instruction and at most one execution of a signal instruction (at the end). Condition queues are now a legitimate part of the shared data store; they must be included in a snapshot and in a monitor restoration. Three aspects of executing such a procedure which are not present when executing a monitor procedure without any wait or signal instructions are: (1) treating segments of a monitor procedure execution as atomic (uninterruptible) monitor operations (with respect to both process interaction and R-chaining); (2) recalling an active process into a condition queue; (3)

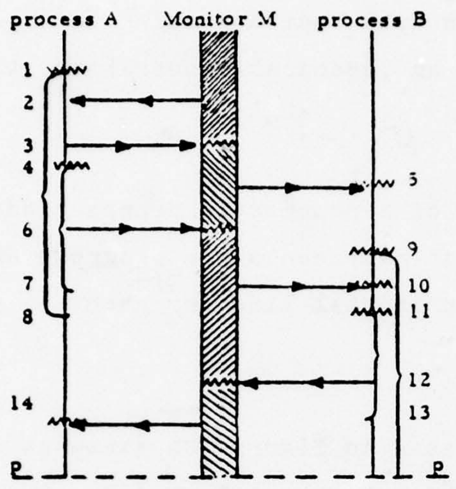


Figure 9c A system history

notifying a waiting process of a rollback.

First, when a process executes a monitor procedure that has wait and signal instructions, it generally goes through an alternating sequence of monitor-possessing periods and waiting periods during which it does not possess the monitor. Only the process activity during each of those monitor-possessing periods is uninterruptible (by other processes) and thus is an atomic monitor operation. As before, an atomic monitor operation is classified as a reference operation, an update operation, or a reference-update operation. Since it is generally impossible to predetermine a sequence of atomic monitor operations involved in execution of such a monitor procedure, a practical approach is to classify the monitor procedure as a whole and then pass that classification to all of its atomic monitor operations as they are dynamically defined. Note that execution of a wait or signal instruction should be treated as an update action since it changes the content of a condition queue that is a part of the shared data store. Analogously, when a waiting process is awakened, the awaking action is treated as a reference action since the awakened process received information (i.e., wakening signal) from the signalling process. In addition, an awakened process checks its mailbox first. Thus the rule that obtaining monitor possession and checking a mailbox form an inseparable operation still holds.

Notation:

(N9) A waiting period of a process is represented by a discontinuity in the vertical line that represents the progress of the process.

(N10) An undirected horizontal line represents a monitor reference-update operation.

For example, process A in Figure 10a executes a monitor procedure (involving wait instructions) from A.3 to A.8. The monitor procedure execution consists of three atomic monitor reference-update operations (A.3, A.6, and A.8). Process A is in a condition queue after A.3 until the beginning of A.6 (at which time it is awakened by the signal generated by B at the end of B.5), and again after A.6 until the beginning of A.8. Note that process A establishes an RP at (the beginning of) A.6. Actually a process may establish multiple RPs during a single monitor procedure execution if the execution involves multiple atomic monitor operations.

Second, restoration of the monitor, which is required when a process rolls back to an RP, may include recalling some processes into the condition queues in which the processes slept previously. For example, assume that process B in Figure 10a fails validation test B.9 and needs to roll back to B.4. Process B is then responsible for restoring the monitor to the state that was recorded at monitor reference-update B.5, which must include restoration of the condition queue that contained process A. Obviously process B can not restore the queue by itself but it can only request (by mail) process A to roll back to RP A.6 and return into the condition queue. When process A has returned into the queue, process B must detect that the monitor

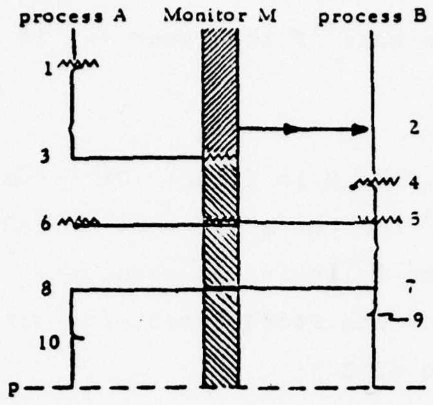


Figure 10a A system history containing waiting periods of a process

store has been completely restored and then roll back to B.4.

To implement this scheme, a monitor store incorporates the following resources transparent to the program designer: a register DOOR, two single-position condition queues MRP and RCPR, and two multiple-position condition queues UNLOCK and PRTR. (Here names MRP, RCPR, and PRTR were derived from "monitor-restoring process", "recalled process", and "processes to roll back out of monitor procedures", respectively.) When DOOR is in a "locked" state, the shared data store can not be accessed by any process except the <u>monitor-restoring process</u> that has locked the store. A process which has obtained monitor possession checks its mailbox but if it finds the shared data store locked when it accesses the store to perform an operation on it, then the process will execute "wait(UNLOCK)". Therefore, queue UNLOCK keeps a set of processes that are waiting for the shared data store to be unlocked. Queue MRP is a single-position queue in which the monitor-restoring process that has already restored the shared data store except condition queues and locked the shared data store may be waiting. A process which has received a return request (from the monitor-restoring process) returns into special queue RCPR and then the monitor-restoring process, solely capable of accessing the locked shared data store including condition queues, transfers the recalled process from RCPR to the destination condition queue w. (The name of the queue w is obtained from the monitor snapshot.)

For example, if process B in Figure 10a becomes responsible for restoring the monitor to the state that existed immediately before B.5, then it performs the following in sequence:
(1) restores the shared data store except for the condition queues by using the snapshot taken at B.5;
(2) learns, by comparing the snapshot with the current state of condition queues, that process A must be returned into a condition queue;
(3) sets DOOR to "locked", thereby locking the shared data store;
(4) puts a recall notice "roll back to RP r and return to special condition queue RCPR" into mailbox MB(A);
(5) executes "wait(MRP)" (thereby releasing the monitor).

Therefore, a process which has received a recall notice performs the following in sequence:
(1) rolls back to an RP;
(2) obtains the monitor possession;
(3) executes a special instruction "signal(MRP)-and-wait(RCPR)" which causes the monitor-restoring process to be awakened from MRP and the recalled process itself to enter RCPR.

The awakened monitor-restoring process performs the following in sequence:
(1) if RCPR is not empty, transfers the recalled process in RCPR into the destination condition queue w, where w is identified from the monitor snapshot;
(2) checks if the condition queues have been completely restored (by comparing the monitor snapshot with the current state of the queues);

(2.1) if so, unlocks the shared data store by changing DOOR into an "open" state, executes "signal(UNLOCK)" (thereby releasing the monitor), and rolls back to an RP to make a retry;

(2.2) otherwise, executes "wait(MRP)".

When a process is ready to exit from a monitor procedure while the shared data store is not locked, the process examines programmer-transparent queue UNLOCK and if it is not empty, executes "signal(UNLOCK)" or else simply exits.

Now suppose a process X decides to restore a monitor to a previous state first and then to roll back to a base RP X.q. Process X obtains the monitor possession and checks its mailbox as usual. If there is a recall notice from process Y asking X to roll back to a branch RP X.r and return into special queue RCPR, then the shared data store has already been locked by process Y and thus process X compares base RP X.q with branch RP X.r. If branch RP X.r was established before base RP X.q then process X will try to roll back to RP X.r and enter into RCPR. Otherwise, process X must reestablish the shared data store to a state preceding the state which Y has been trying to reestablish. Process Y is now asked to roll back to the RP established when it became R-chained to RP X.q. Therefore, after learning that base RP X.q was established before branch RP X.r, process X performs the following in sequence:

(1) obtains the monitor possession;

(2) leaves rollback notices into appropriate mailboxes including that of process Y;

(3) determines whether or not there is a process that X has to recall into RCPR;

(3.1) if there is, executes "signal(MRP)" so that process Y may be awakened to receive the rollback notice while process X rolls back to an RP to make a retry;

(3.2) if there is not, executes a special instruction "replace(MRP)" which causes the process Y in queue MRP to be replaced by the process X executing the instruction. As a consequence, process Y takes possession of the monitor. Once the process Y in MRP is awakened, it will read the rollback notice.

Third, restoration of a monitor may include driving processes out of some condition queues. For example, assume that process B in Figure 10b has failed validation test B.p. At this point process A is sleeping in a condition queue and thus will not check its mailbox unless it is awakened. Process B should therefore waken process A so that process A may read the rollback notice. The programmer-transparent multiple-position queue PRTR in a monitor store is used to buffer the processes that have been waiting in condition queues and need to be awakened to read rollback notices. When an RBE e of process X fails, process X accesses monitor store MS(M) and examines RPQ(M,e). Then process X performs the following in sequence:
(1) puts rollback notices in the mailboxes of all the processes that established the RPs contained in the RP queue.
(2) moves each of those processes that established RPs contained in the RP queue and have been sleeping in condition queues, into special queue PRTR by executing a special instruction "move-to-PRTR(condition-queue-name)";
(3) locks the shared data store;
(4) executes a special instruction "signal(PRTR)-and-wait(MRP)" which causes a process Y at the front of queue PRTR to be awakened while process X enters into queue MRP. The awakened process Y then reads the rollback notice.

The actions taken by a process while releasing a monitor without executing a signal instruction are summarized as follows.

<u>Procedure</u>: (P1) "Monitor exit"
(case 1) DOOR = "open":

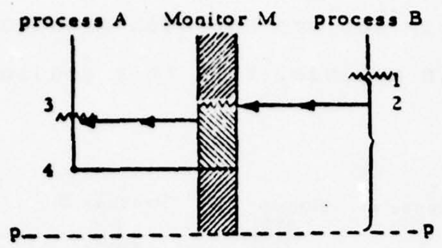


Figure 10b A process (A) that needs to be driven out of a queue

```
        if UNLOCK is empty
            then simply release the monitor;
            else execute "signal(UNLOCK)";
    (case 2) DOOR = "locked":
        if PRTR is empty
            then execute "signal(MRP)";
            else execute "signal(PRTR)";
    End-Procedure
```

Clearly the message interpretation procedure Pb (in Appendix B) must now be extended to handle the situations discussed in this section. The extended version will not be detailed in this paper.

3.6 Indirect recovery

We now consider the case in which a process that has produced erroneous information for other processes cannot have an opportunity to execute its own validation test. The system would crash unless additional features have been incorporated. In Figure 11, assume that process A stores erroneous information into the monitor store at A.3. Process B takes the erroneous information at B.4 and subsequently fails the validation test at B.p. A retry by process B might fail again at B.p since the erroneous information generated at A.3 is still in the monitor and is referenced again. Meanwhile process A is not aware of the failure(s) of process B and is waiting inside a condition queue for a wakening signal that has to be supplied by process B (after B.p) but has not been produced because process B has not passed B.p. In a sense, this is a deadlock situation.

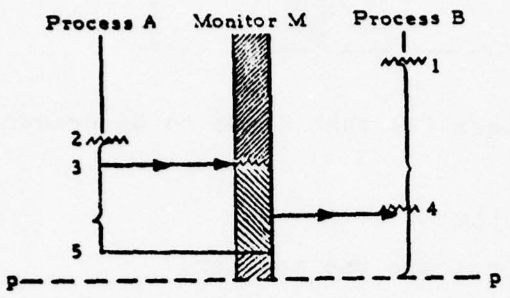


Figure 11 A process (A) entering a condition queue after depositing bad information

If process B does not maintain any RP earlier than B.1, then process B will have to be abandoned after the unsuccessful retries with all the available alternates and the system will crash. In this case, the system crashes despite the possibility that validation test of RBE [A.2:] may be able to detect the error and a retry by process A from A.2 may circumvent the error. That is, the resources in the system are not fully utilized.

There are two possible solutions to this problem. One is to make the programmer-transparent watchdog process (introduced in section 3.3.1) also responsible for periodically examining the status of every process and, upon detection of a process that has spent an excessive amount of time in a condition queue, forcing the process to roll back to the immediately preceding base RP and to begin a retry. A process X that has exhausted all the alternates waits (at the point of the last failure) until another process Y that interacted with X and has slept long, is forced by the watchdog process to roll back. Process Y sends out the notice of this rollback and, upon (being awakened by the watchdog process and) receiving the notice, process X initiates another retry. The other solution is to allow S-propagation as a last resort when process X has exhausted all the alternates and is about to be abandoned. That is, process X requests the processes that have produced information which was referenced by X, to roll back to their earlier base RPs, and thereafter process X makes a retry. This exception (i.e., S-propagation) can be justified since the system would have crashed anyway. However, the first approach of relying upon the watchdog process may allow faster recovery in many cases. Either way, the error is indirectly detected and recovered.

Suppose now that process B in Figure 11 maintains an RP B.0 preceding B.1 and its rollback to B.0 entails the rollback of process A to RP A.a preceding A.2 (through R-propagation). Assume, as before, that process A stores erroneous information at A.3. As process B repeatedly fails at B.p and rolls back to B.0 (once every n failures at B.p, where n denotes the number of alternates within the RB originally entered at B.1), process A will repeatedly roll back to A.a and reenter the RB originally entered at A.2. Again this looping situation cannot be circumvented unless process A executes a new alternate, starting at A.2. Therefore, it is desirable for a process to try a new alternate at some time if it is repeatedly forced to reenter the same RB.

4. Fault-tolerant monitors and extended recovery caches in systems containing multiple monitors

This section discusses the problems introduced by the systems containing more than one monitor and solutions to those problems. Systems without nested monitors are considered in section 4.1 and then systems with nested monitors are considered in section 4.2.

4.1 Systems containing multiple non-nested monitors

If a process can access multiple monitors, then the process must in general maintain snapshots of multiple monitors and its rollback may involve restoration of multiple monitors. Process B in Figure 12a is such a process. An RBE [B.a:] of process B may become a potential recaller of both M1 and M2 and through them, a potential recaller of A and C. The extension of both a fault-tolerant monitor and an extended recovery cache to handle this is obvious and thus is not elaborated.

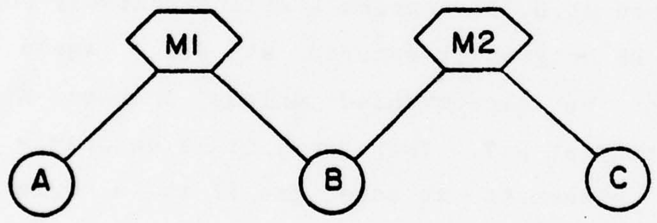


Figure 12a A system of three processes and two non-nested monitors

A less obvious problem is that of handling a rollback chain formed through multiple monitors. For example, a rollback chain may be established in Figure 12b which connects RBE [A.1:] of process A to process C (at C.5) through three other system components: monitor M1 (at A.2), process B (at B.3), and monitor M2 (at B.4). In that case, rollback of process A to RP A.1 will require rollback of process B to RP B.3 which will in turn require rollback of process C to RP

C.5. A natural path through which a rollback notice or validation notice is sent is the rollback chain itself. Therefore, when process B is about to excute an update at B.4 that R-chains RBE [A.1:] to M2, it must detect that the monitor (M1) through which it became R-chained to RBE [A.1:] is different from the monitor (M2) which it currently R-chains the RBE to, and must record this fact in cache segment CS(B.3). Process B will then be responsible for relaying a rollback notice or validation notice (related to [A.1:]) to the processes that may become R-chained to [A.1:] through M2. If process B is asked to roll back to B.3, it can perform the following in sequence:

(1) finds from CS(B.3) the record of R-chaining [A.1:] to M2;

(2) obtains possession of M2 and restores the shared data store;

(3) finds RP C.5 from RPQ(M2,[A.1:]) and puts a rollback notice in the mailbox of process C;

(4) releases M2 and rolls back to B.3.

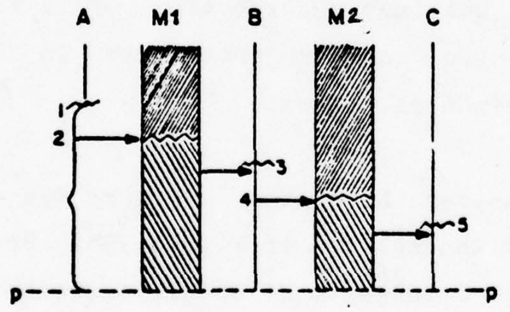


Figure 12b A rollback chain connecting five system components

A notice of the validation of [A.1:] can be relayed in an analogous manner. In short, if an RBE [X.a:] of a process X has become a potential recaller of a monitor M through an update by another process Y, then process Y serves as a messenger or acting agent of RBE [X.a:].

In systems using multiple monitors, the function of a watchdog process needs to be extended further. To see the need, consider a process X waiting in a queue within monitor store MS(M2) while another process Y puts a recall notice in the mailbox of process X within another monitor store MS(M1) and enters queue MRP. If process X is waiting for a wakening signal from process Y, then both processes are deadlocked until the special watchdog process intervenes. The watchdog process now has to waken process X from the condition queue and then force it to release M2 and check the mailbox MB(X,M1).

4.2 Systems containing nested monitors

If a monitor procedure p includes calls to other (nested) monitor procedures, then execution of p is treated as a compound monitor operation which may be broken into several atomic operations between which RPs may be established.

Notation: (N10) A continuous execution of a monitor procedure including calls to nested monitor procedures is represented by a vertically shaded horizontal column.

For example, consider the system in Figure 13a of which a segment of execution history is depicted in Figure 13b. Process A executes a monitor procedure that belongs to M1 and involves both a reference to and an update of M1, from A.3 to A.7. Similarly, process C executes a compound monitor operation from C.2 to C.8. On the other hand, the monitor operation C.9 is an atomic monitor operation since the monitor procedure (of M2) executed does not involve either a call to M3 nor an execution of wait or signal instruction. For the same reason, every execution of a procedure of M3 shown in the figure is an atomic monitor operation. RBE [A.1:] becomes a potential recaller of M3 at A.4, and the snapshot of M3 taken at this time is appended to the snapshot of M1 taken at A.3. Process C then becomes R-chained to RP

A.1 at C.5 and thus an RP is established. This RP establishment involves checkpointing of the state of both process store PS(C) and monitor store MS(M2) (i.e., creation of a cache segment in PS(C) and taking a snapshot of MS(M2)).

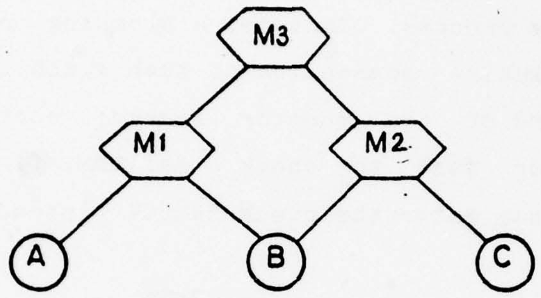


Figure 13a A system containing a nested monitor M3

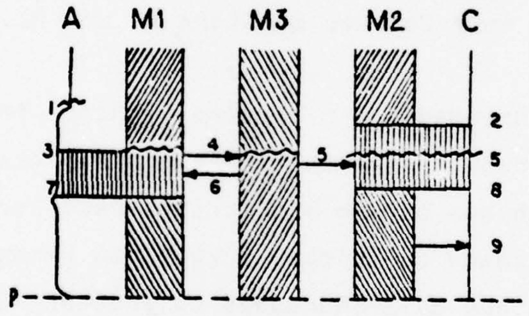


Figure 13b A history of the system in 13a

Given that taking a snapshot of a nested monitor and establishment of an RP are performed as mentioned above, another area that requires an extension is communication of rollback notices or validation notices among processes. A message communication scheme is given below.

First, when a process in possession of several nested monitors executes wait(q) to enter a condition queue q in the last acquired monitor Mn, it releases only Mn but it maintains the possession of other earlier acquired monitors while sleeping in q. Thus a sleeping process may be in possession of some monitors. In order to allow the mail kept in the mailboxes within such a monitor (possessed by a sleeping process) to reach the receiving processes, the mailboxes must be accessible to any process. That is, a sleeping process does not have complete exclusive possession of such a monitor but only keeps the shared data store of the monitor locked; another process may acquire the monitor just to check a mailbox, find the shared data store locked, and thus enter the queue UNLOCK (introduced in section 3.5).

Second, a nested monitor maintains mailboxes only for system components (i.e., processes or monitors) that have direct access capabilities to the monitor. For example, monitor M3 in Figure 13a maintains mailboxes only for two monitors M1 and M2.

Third, mail addressed to a process which is R-chained to a monitor M is forwarded through the R-chain. For example, if process C in Figure 13a, which has become a potential recaller of M3 through M2, needs to send a message to process B which is R-chained to M3 through M1, then it can put the mail addressed to process B in the mailbox within M3 <u>maintained for M1</u>. In a sense, M1 is viewed as a <u>cover</u> of process A or B when either process accesses M3 through M1. This mail can reach process B in two ways. First, if process B accesses monitor M3 through M1 before process A, it checks the mailbox for its cover M1. Since the mailbox is not empty, process B picks up the mail, releases M3, and then opens the mail. Process B learns that the mail is addressed to itself and thus releases M1 before starting the interpretation of the mail. In the other case where process A accesses M3 through M1 before process B, it takes the mail (addressed to B) from the mailbox within M3 maintained for its cover M1, releases M3, and puts the mail into the mailbox MB(B,M1). Process A reaccesses M3 and the mail will then be read later when process B accesses M1. Therefore, both processes A and B are responsible for checking the mailbox within M3 maintained for their common cover M1.

The above scheme implies: when a process X in an RBE that is a potential recaller of a nested monitor M decides to send a rollback or validation notice to another process Y that is R-chained to M, process X must know the R-chain leading to Y. Process X must use the R-chain as the destination address of the mail. To enable this, when a process becomes R-chained to a nested monitor M and establishes a branch RP, the name of the RP is prefixed by the name of the cover and then the prefixed RP name is inserted into the RP queue attached to table PRST(M). For example, when process C in Figure 13b becomes R-chained to M3 at C.5, the prefixed RP name M2$C.5 is inserted into RPQ(M3,[A.1:]). If a monitor M is a multi-level nested monitor, then the RP name will be prefixed by the names of all the monitors on the R-chain being established.

5. Shared resources not contained within a monitor

Enclosing some shared resources within a monitor may lead to an inefficient system because of the rule that a monitor may have no more than one of its procedures in execution at a time. Each use of such a resource by a process generally takes a large amount of time and thus there is strong motivation for allowing several processes to use the resource simultaneously as long as there is no danger. For example, it is sometimes safe to allow several processes to simultaneously reference the same sizable record (without modification). An efficient design would then provide a monitor in which processes, one at a time, obtain or return only access rights to a shared resource while the shared resource itself is located outside the monitor and may be used by several processes possessing access rights [H2]. Figure 14a depicts such a design and Figure 14b shows a typical structure of a program using a shared resource associated with but located outside a monitor. Program structuring in the form shown in Figure 14b is not a requirement, however.

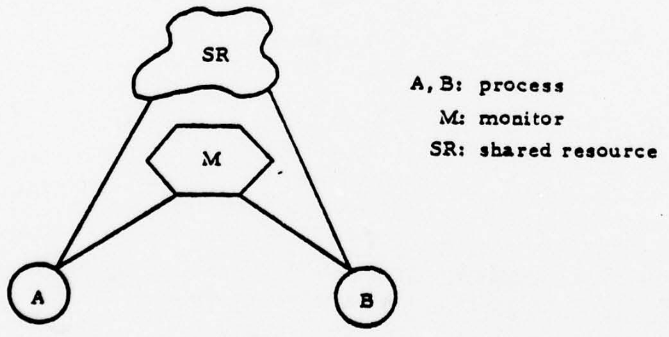


Figure 14a A shared resource SR associated with but located outside monitor M

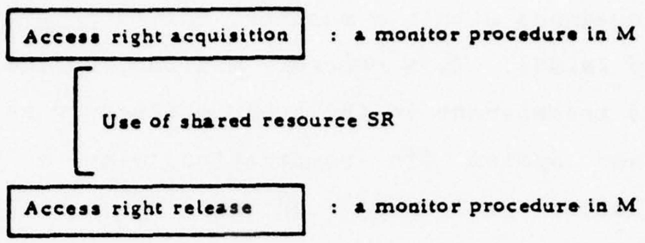

Access right acquisition
: a monitor procedure in M
Use of shared resource SR
Access right release
: a monitor procedure in M

Figure 14b Typical program structure for using SR in 14a

We assume that the portions (e.g., a statement or a block) of a program that uses a shared resource outside a monitor can be recognized through automated analysis of the program text; this is a prerequisite for applying the programmer-transparent coordination approach discussed so far to systems containing such a shared resource. A syntactic unit recognized as one using such a shared resource may be a statement, a block of statements, or the entire portion between the monitor procedure call for obtaining an access right and the other for returning the access right (depicted in Figure 14b). Execution of such a syntactic unit is treated as a unit operation on a shared resource and may involve reference, update, or both; it is analogous to the execution of a monitor procedure. Once a process obtains an access right to a shared resource SR, it may execute many unit operations on SR (each involving reference, update, or both) before returning the access right. Recovery of processes sharing a resource SR outside a monitor can be facilitated again by maintaining table PRST(SR) and operating mailboxes for the processes that may access SR, in a manner similar to that of managing table PRST and mailboxes within a monitor store. The only problem is the possibility of conflicts among several processes that possess access rights to SR and try to update table PRST(SR) or mailbox simultaneously.

A simple and natural solution is to enclose table PRST(SR) (together with the attached RP queues), mailboxes, and their management procedures within a monitor, called <u>recovery monitor (RM)</u> and denoted by RM(SR). This recovery monitor, unlike the monitor M in Figure 14b, is transparent to the program designer and can be provided by the processor system (in cooperation with a parallel program compiler). Before performing a unit operation on the resource SR, a process always accesses recovery monitor RM(SR) to examine table PRST(SR). Acquisition of recovery monitor RM(SR) and the subsequent mail check form a single inseparable operation as before. Although multiple processes may possess access rights to SR simultaneously, they can become potential recallers of SR or become R-chained to RBEs through SR one at a time only.

If the number of resources shared outside a monitor is large, then the number of recovery monitors provided will also be large. If a smaller number of recovery monitors is desired, then a recovery monitor can be provided for and associated with a group of shared resources as if the group of resources were a single large shared resource; the price paid is the increased frequency of process synchronization due to increased access to the same recovery monitor.

6. Summary

Programmer-transparent coordination of process rollbacks in systems of interacting fault-tolerant processes is a goal pursued in this paper. The approach explored is based on the strategy of prohibiting rollback propagation due to suspicion, in keeping with the spirit of designing processes to harmoniously cooperate with one another. An important requirement that any practical scheme for automated coordination must meet is the maintenance of a manageable number of RPs at all times. Two useful rules were formulated to meet the requirement: maintenance of minimal RP sets based on PRSs, and safe discard of useful RPs under practical constraints such as limited memory space. A monitor mechanization, termed fault-tolerant monitor, and an extension of recovery cache were presented to demonstrate the viability of the automated coordination approach.

There is a trade-off between automated coordination approaches and programmed coordination approaches. Automated coordination involves greater time and space overhead than coordination by the program designer. On the other hand, rollback coordination, even with the help of language tools such as a conversation [R1], can often become an unbearable burden on the program designer. This limits programmed coordination to being practical only when it is applied at a relatively macroscopic level. In fact, an optimal approach in some situations may be a combination of programmed coordination and automated coordination approaches such that the program designer is concerned only with coordination at a more macroscopic level while coordination at a microscopic level is automatically handled by an intelligent underlying processor system equipped with fault-tolerant monitors and extended recovery caches. Such a combination as well as evaluation of the performance of an intelligent processor system presented in this paper remains as a subject for future study.

References

[A1] Anderson, T. and Kerr, R., "Recovery blocks in action: a system supporting high reliability", Proc. 2nd Int'l Conf. on Software Engineering, 1976, pp.447-457.

[B1] Brinch Hansen, P., "The programming language Concurrent Pascal", IEEE Trans. on Software Engr., June 1975, pp.199-207.

[C1] Chandy, K.M., "A survey of analytic models of rollback and recovery strategies", Computer, May 1975, pp.40-47.

[D1] Dahl, O., Dijkstra, E.D., and Hoare, C.A.R., Structured Programming, Academic Press, 1972.

[D2] Dijkstra, E.W., "Hierarchical ordering of sequential processes", Acta Informatica, Vol.1, No.2, 1971, pp.115-138.

[H1] Hecht, H., "Fault-tolerant software for spacecraft applications", Tech. Rept. SAMSO-76-40, Aerospace Corp., Dec. 1975. (also in [Y2])

[H2] Hoare, C.A.R., "Monitors: an operating system structuring concept", Comm. of ACM, Oct. 1974, pp.549-557.

[H3] Horning, J.J., Lauer, H.C., Melliar-Smith, P.M., and Randell, B., "A program structure for error detection and recovery", Lecture Notes in Comp. Sci., vol. 16, Springer-Verlag, 1974, pp.171-187.

[K1] Kim, K.H. and Ramamoorthy, C.V., "Recent developments in software fault tolerance through program redundancy", Proc. 10th Hawaii Int'l Conf. on System Sciences, Jan. 1977, pp.234-238.

[M1] Meyers, M.N., Routt, W.A., and Yoder, K.W., "Maintenance software", Bell System Technical Journal, Special Issue on No.4 ESS, Sept. 1977, pp.1139-1167.

[P1] Parnas, D.L. and Wurges, H., "Response to undesired events in software systems", Proc. 2nd Int'l Conf. on Software Engineering, 1976, pp.437-446.

[R1] Randell, B., "System structure for software fault tolerance", IEEE Trans. on Software Engr., June 1975, pp.220-232.

[R2] Repton, C.S., "Reliability assurance for System 250: a reliable, real-time control system", Proc. Int'l Computer Communication Conf., 1972, pp.297-305.

[R3] Russell, D.L., "Process backup in producer-consumer systems", Proc. 6th Symp. on Operatings Systems Principles, Nov. 1977, pp.151-157.

[S1] Shrivastava, S.K. and Banatre, J.P., "Reliable resource allocation between unreliable processes", Tech. Rept. SRM/177, Computing Lab., Univ. of Newcastle upon Tyne, (to be published in IEEE Trans. on Software Engineering).

[W1] Wulf, W.A., "Reliable hardware-software architecture", IEEE Trans. on Software Engr., June 1975, pp.233-240.

[Y1] Yau, S.S. and Cheung, R.C., "Design of self-checking software", Proc. 1975 Int'l Conf. on Reliable Software, pp.450-457.

[Y2] Yeh, R.T. ed., 'Special issue on fault-tolerant software', Computing Surveys, Vol.8, No.4, December 1976.

Appendix A: Proofs of two lemmas stated in section 3

Proof of L1: To each current potential recaller e of a process there corresponds a valid RP r established by the process when RBE e became a potential recaller of the process. The RP r is obviously the most recent valid execution point of the process when RBE e has failed. On the other hand, to each RP r currently maintained by a process there correspond one or more current potential recallers of the process. Since the process needs to roll back when and only when any of its potential recallers fails, the process maintains no redundant RPs. Therefore, processes maintain the minimum number of RPs required to make every rollback with minimum distance. Q.E.D.

Proof of L2: Suppose that the members of PR*([Z.a:]) known to process Z at time t are all partially validated RBEs while there is an on-going RBE [X.e:] belonging to PR*([Z.a:]) but not known to process Z. Obviously RBE [X.e:] is an indirect potential recaller of RBE [Z.a:] and thus there must be a sequence of RBEs $(e_1,e_2,---,e_n)$ where e_1 = [X.e:], e_n = [Z.a:], and e_i $(1 \leq i<n)$ is a potential recaller of e_{i+1}. Suppose e_k, where $1 \leq k<n$, is known to process Z at t while e_{k-1} is not. The notice of the partial validation of e_k must have included information on the then known members of PR*(e_k) that includes e_{k-1}. In addition, when process Z received the notice, it must have learned both the fact that e_k had been partially validated and the fact that e_{k-1} was a member of PR*(e_k). Thus every e_i, $1 \leq i \leq n$, must be a member of PR*([Z.a:]) known to process Z at t. This means that e_1 (= [X.e:]) is an on-going RBE known to process Z, in contradiction to the assumption made at the beginning. Q.E.D.

Appendix B: A message interpretation procedure in systems using one monitor without wait or signal instructions

Procedure: (Pb) "Message interpretation (Y: process; M: monitor)"

{Intended-monitor-operation = The monitor operation that process Y would have performed if the mailbox, that contained the messages currently being interpreted, had been empty}

{Operation A = a monitor update or reference}

{Operation B = notification of a success of the validation test of [Y.c:]}

{Operation C = notification of the discard of RBE [Y.c:]}

{Operation D = notification of the discard of a useful base RP}

{Operation E = notification of new members of the PR* of already partially validated RBE [Y.c:]}

{Message 1 = "RBE [X.a:] that is a potential recaller of RP Y.r has been completely validated"}

{Message 2 = "RBE [X.a:] that is a potential recaller of RP Y.r has been partially validated, and the currently known PR*([X.a:]) is ---"}

{Message 3 = "RBE [X.a:] has failed, so roll back to RP Y.r"}

{Message 4 = "base RP X.j has become a defunct branch of base RP X.i and RBE [X.j:] is no longer a potential recaller of RP Y.r"}

{Message 5 = "RBE [W.a:] that is a previously known member of PR*([X.b:]), where RBE [X.b:] is a potential recaller of RP Y.r, has been partially validated and the new members of PR*([X.b:]) are ---"}

sort the messages in the chronological order of the RPs mentioned in the messages;

discard the messages placed after the first message of type Message 3 in the sorted list;

starting with the beginning of the sorted list, interpret each message as follows:

```
(case 1) Message 1:
  (case 1.1) Intended-monitor-operation = A, B, D, or E:
    if RP Y.r remains then
    begin
       remove [X.a:] from table PRST(Y) and the record of PRS(Y.r);
       if PRS(Y.r) is now empty then
       begin
          discard RP Y.r;
          if discard of RP Y.r has caused a base RP Y.q to have no
             branch RPs to support and RBE [Y.q:] has passed its vali-
             dation test, then RBE [Y.q:] has become completely vali-
             dated, so add its notification to the list of intended
             monitor operations (i.e., operations to be performed
             within the monitor store)
       end
    end;
    "continue" (i.e., if there is another message, interpret it;
       otherwise, regain the monitor);
  (case 1.2) Intended-monitor-operation = C:
    if RP Y.r precedes RP Y.c
       then do the same as in (case 1.1)
       else do nothing and continue;
(case 2) Message 2:
  (case 2.1) Intended-monitor-operation = A, B, D, or E:
    update the status of [X.a:] in both table PRST(Y) and the
       record of PRS(Y.r);
    if there is a base RP Y.q supporting RP Y.r then
    begin
       update the record on PR*([Y.q:]);
       if RBE [Y.q:] has been partially validated and  all the members
       of PR*([Y.q:]) now have the "partially validated" status,
         then RBE [Y.q:] has become completely validated, so
            begin
```

```
                    discard base RP Y.q and the branch RPs including Y.r
                      supported by Y.q;
                    add the notification of the decease of both RBE [Y.q:]
                      and other RBEs that have become completely validated
                      together with [Y.q:], to the list of intended
                      monitor operations
                  end
                else add the notification of the new members of PR*([Y.q:])
                  and of the partial validation of [X.a:] to the list of
                  intended monitor operations
            end;
            continue;
          (case 2.2) Intended-monitor-operation = C:
            if RP Y.r precedes RP Y.c
              then do the same as in (case 2.1)
              else do nothing and continue;
        (case 3) Message 3:
          abolish the list of intended monitor operations and roll back to Y.r;
        (case 4) Message 4:
          remove RBE [X.j:] from table PRST(Y) and the record of PRS(Y.r);
          continue;
        (case 5) Message 5:
          (case 5.1) Intended-monitor-operation = A, B, D, or E:
            do the same as in (case 2.1) except RBE [W.a:] substituting for
              [X.a:];
          (case 5.2) Intended-monitor-operation = C:
            if RP Y.r precedes RP Y.c
              then do the same as in (case 5.1)
              else do nothing and continue;

        End-Procedure
```

Acknowledgement

The authors wish to thank C. Hasselbach and L. Simoncini for helpful discussions during the research reported in part I, F. Farrand for helpful discussions on the work reported in part III, F. P. Dyke for his guidance during the initial period of this project, and H. Hecht for his suggestions and guidance at every stage of this project.