An Annual Report
Grant No. NAG-1-260
March 5, 1982 - December 31, 1986

THE IMPLEMENTATION AND USE OF ADA ON DISTRIBUTED SYSTEMS WITH HIGH RELIABILITY REQUIREMENTS

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665

Attention: Mr. Michael Holloway
ISD, MS 478

Submitted by:

J. C. Knight
Associate Professor

Report No. UVA/528213/CS87/110
March 1987

SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA 22901

An Annual Report
Grant No. NAG-1-260
March 5, 1982 - December 31, 1986

THE IMPLEMENTATION AND USE OF ADA ON DISTRIBUTED SYSTEMS WITH HIGH RELIABILITY REQUIREMENTS

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: Mr. Michael Holloway
ISD, MS 478

Submitted by:

J. C. Knight
Associate Professor

Department of Computer Science
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/528213/CS87/110
March 1987

Copy No. 4

TABLE OF CONTENTS

OVERVIEW

The purpose of this grant is to investigate the use and implementation of Ada* in distributed environments in which reliability is the primary concern. In particular, we are concerned with the possibility that a distributed system may be programmed entirely in Ada so that the individual tasks of the system are unconcerned with which processors they are executing on, and that failures may occur in the software or underlying hardware.

Over the next decade, it is expected that many aerospace systems will use Ada as the primary implementation language. This is a logical choice because the language has been designed for embedded systems. Also, Ada has received such great care in its design and implementation that it is unlikely that there will be any practical alternative in selecting a programming language for embedded software.

As computers become increasingly powerful, traditional arguments favoring the construction of locally distributed systems become less convincing. If one's only aim in distributing a system over multiple computers is to acquire more computing power, one should purchase a bigger computer. The savings in software complexity, synchronization overhead, and data transfer overhead make powerful single-computer systems attractive. Additionally, distributed systems are only as good as their weakest nodes. If, for example, each of three computers in a system will operate correctly 99 percent of the time, the system as a whole will function, on average, only 97 percent of the time. To address this last problem, researchers are inventing techniques which allow software to tolerate the failure of one or more nodes of a distributed system. Success in this endeavor will make these systems *more* reliable than their undistributed cousins. Because numerous computer systems in such areas as the defense and aerospace industries must be highly reliable, this research has proceeded apace.

* Ada is a trademark of the U.S. Department of Defense

In previous work under this grant we have shown the general inadequacy of Ada for programming systems that must survive processor loss. We have also proposed a solution to the problem in which there are no syntactic changes to Ada. While we feel confident that this solution is adequate, we cannot be sure until the solution is tested. A major goal of the current grant reporting period therefore is to evaluate the approach using a full-scale, realistic application. The application we are using is the Advanced Transport Operating System (ATOPS), an experimental computer control system developed at NASA Langley for a modified Boeing 737 aircraft. The ATOPS system is a full authority, real-time avionics system providing a large variety of advanced features.

We have also shown previously under this grant that Ada makes no provision for software fault tolerance. We consider it to be important that attention be paid to software fault tolerance as well as hardware fault tolerance. The reliability of a system depends on the correct operation of the software as well as the hardware. A second major goal of the current grant reporting period is to extend previous work in new methods of building fault tolerance into concurrent systems.

During this grant reporting period our primary activities have been:

(1) Determination of a set of criteria by which the proposed method will be judged.

(2) Extensive interaction with personnel from Computer Sciences Corporation and NASA Langley to determine the requirements of the ATOPS software.

(3) A preliminary analysis of the Ada implementation of a skeleton ATOPS system.

(4) Preparation of a report summarizing the state of the art in backward error recovery in concurrent systems.

The various documents that have been prepared in support of this research during the grant reporting period are included in this report as appendices. The criteria we have determined for evaluation of the approach to fault tolerance are contained in appendix 1. A preliminary version of the requirements for the ATOPS system is contained in appendix 2. This requirements specification is not a formal document in which detailed specifications are included. Rather it is a description of certain aspects of the ATOPS system at a level of detail that best suits the needs of the research. The results of the preliminary analysis are included in appendix 3. This appendix is in the form of a paper that will be presented at an upcoming Ada conference. Appendix 4 contains our survey of backward error recovery techniques. This survey will appear in the text "Resilient Computing Systems, Volume 2" edited by T. Anderson, published by John Wiley. A list of papers and reports prepared under this grant, other than the annual and semi-annual progress reports, is presented in Appendix 5. For the most part, section and figure numbers as they will appear in the final form of each document have been retained in the appendices.

APPENDIX 1

EVALUATION CRITERIA

EVALUATION CRITERIA

In order to assess the quality of the proposed solution to Ada's limitations on distributed targets, a number of evaluation criteria are outlined in this section. These criteria are intended to be general standards by which any non-transparent approach can be judged in the context of any application requiring tolerance of processor failure. They will be applied to the software we develop in response to the ATOPS requirements specification.

Structure

The resulting program should be modular, comprehensible, and modifiable. The extent to which the program fails stylistically is of great concern, since the chosen fault-tolerance approach places much of the burden of recovery on the programmer, and the effect of that onus on a realistic program structure is unknown at this time. In order for non-transparent fault tolerance to be feasible for Ada programs, those programs must be well-structured.

This criterion is particularly subjective, but we will decide if the goal is met by determining whether or not information hiding and object isolation techniques can be used effectively for a particular application.

Dynamic Service Degradation

A distributed program intended to survive hardware failures must match the processing load following a failure with the processing capacity remaining. One way to avoid the waste inherent in adding extra computing power to the original system is to specify degraded or *safe* service to be provided following failure. If the application is amenable to such a specification, the non-

transparent approach can allow a high percentage of the overall computing power of the system to be used at all times. In contrast, the transparent approach normally must be only lightly loaded so as to be able to provide identical service before and after the loss of an entire processor.

The suitability of an application for provision of safe service following hardware failure should be determined during the specification process. Experts should be consulted, and their knowledge about the particular application should be incorporated into the requirements document.

Task Distribution Flexibility

At a certain point in the design process, task redistribution may require massive code revision. If this point in time is early in the design process, design-time flexibility will be drastically reduced by the non-transparent approach. Additionally, ordinary software maintenance should not be prohibitively expensive when it involves task redistribution. In sum, flexibility of distribution of tasks among the processors should not be overly affected by those aspects of the program necessary for fault-tolerance.

Distribution flexibility should be evaluated upon the completions of both design and coding. Flexibility can be determined by measuring the percentage of the relevant product, the design document or the code itself, which requires modification in order to accommodate task redistribution.

Efficiency

The percentage of the total system computing power devoted to execution of statements necessary for fault-tolerance should be reasonably small. The memory and time overhead for both the source program and the underlying support system should be considered. The overhead due to fault-tolerance is categorized in the following sections:

Entry Call Renaming

Since a replacement for a lost task cannot be given the same name, application program code must sometimes be embedded in the normal program algorithms when calls to entries of tasks located on other machines are involved. The following code can be omitted only when the calling task is always aborted upon failure of the machine which runs the receiving task:

```
if beforeFailure then
   Perform normal pre-rendezvous computations;
   SoonToBeDeadTask.EntryCall;
   Perform normal post-rendezvous computations;
else
   Perform alternate pre-rendezvous computations;
   ReplacementTask.EntryCall;
   Perform alternate post-rendezvous computations;
end if;
```

Exception Handlers

An exception handler must be associated with each task which calls an entry in a task which runs on a different machine. These handlers do not use CPU cycles during normal execution, but they do add bulk to the program.

Data Management

Data distribution tasks must exist on each machine to provide the programmer with the means to generate consistent copies of critical data items on all machines. Embedded in each task, calls to the data distribution task will increase the complexity of the algorithms and slow their execution. The resulting increase in bus traffic also will burden the system. The key parameters here are the number of data items required for failure survival and the frequency at which they must be recorded on the other machines.

Reconfiguration

Tasks existing on each machine will be responsible for reconfiguration of the system. They will use CPU cycles only at recovery time, but the sheer volume of reconfiguration work required during that critical period may be so large as to cause failure of the system. Most applications which are desired to be hardware fault-tolerant have critical real-time requirements. If the services to be provided after failure are radically different from those offered before failure, the aborting of unwanted tasks and the initialization of newly desired tasks could be prohibitively time-consuming.

This is not a simple issue. Reconfiguration tasks do not necessarily have to complete before application processing resumes. For example, the high-priority reconfiguration task could terminate after performing only critical operations and start a low priority task which eventually would put the entire system in perfect order without disrupting critical processing.

Failure Detection

In order to detect failure when it occurs, a small constant overhead must be paid to produce heartbeats and to listen for those of the other machines. If an implicit token-passing protocol is used for inter-processor communication, failure detection costs nothing, so we will ignore this source of overhead in our analysis.

Message Logging

In order to assess the damage to tasks not on the failed machine, all program-level communications will be recorded. The overhead from this source will vary with the message traffic between machines, so, *ceteris paribus*, inter-processor communication should be minimized.

Alternate Service Casing

Code *might* be embedded in the normal program algorithms in order to be provide alternate service after hardware failure:

```
if beforeFailure then
  Perform normal computations;
else
  Perform alternate computations;
end if;
```

Complexity

The complexity of the software *could* increase geometrically as the number of tolerable hardware failures in the system increases. As an example of this case proliferation consider a structure based on a four processor system that provides different services depending upon which machines are operational. The following code is written from the perspective of machine 1:

```
case systemStatus is
  when All4Up =>
    Process normally;
  when 134Up =>
    Process with machine 2 down;
  when 124Up =>
    Process with machine 3 down;
  when 123Up =>
    Process with machine 4 down;
  when 14Up =>
    Process with machines 2 and 3 down;
  when 13Up =>
    Process with machines 2 and 4 down;
  when 12Up =>
    Process with machines 3 and 4 down;
  when 1Up =>
    Process with machines 2, 3, and 4 down;
end case;
```

The target architecture and the complexity of failure response, then, are critical application-dependent variables in this evaluation process.

APPENDIX 2

ATOPS SYSTEM REQUIREMENTS

REQUIREMENTS OVERVIEW OF ADVANCED TRANSPORT OPERATING SYSTEM SOFTWARE FOR DISTRIBUTED EXECUTION†

John C. Knight Marc E. Rouleau

Department of Computer Science
University of Virginia
Charlottesville, Virginia, 22903

SUMMARY

This document is a high-level software requirements specification for an aircraft computing system capable of performing all flight tasks from take-off through landing. These general requirements have been abstracted from those for the Advanced Transport OPerating System (ATOPS), which is being managed by Computer Sciences Corporation for NASA. Implementations of this specification are not intended to be adequate for any real system; instead, we intend only for the *structures* of resulting programs to be comparable to those of actual aircraft computing systems, Low-level computations will be omitted wherever their absence will not result in a simplification of the overall program structure. This document, then, neglects numerous areas which should be addressed by any comprehensive aircraft computing system software specification. This work has been done in order to provide computer scientists with a realistic model against which they can test hypotheses about the construction of embedded, distributed systems which must perform in real time.

† This work was supported in part by NASA grant NAG1-260

1. INTRODUCTION

The final test of any theory of software construction is practice. Does it work? Unfortunately, most software engineering theorists do not have access to realistic test cases. Previous research by Knight and Urquhart, for example, involves the suitability of Ada† for the programming of embedded, distributed, real-time systems which *must not* fail. In particular, they were concerned with the survival of distributed Ada processes after failure of a nodes running part of the process. A theory about the construction of such systems was proposed. The feasibility of that theory was tested by constructing a distributed testbed which implemented the proposed semantic enhancements to Ada. Indeed, simple Ada programs, *written according to certain guidelines*, can be run on the testbed, and they do survive arbitrary failure of one of the nodes.

The above sequence of research is familiar to all scientists. A problem is identified, a solution is proposed, and the solution is tested. In this case, the success of the toy program tests was encouraging, but many important questions remained unanswered. How do the aforementioned guidelines affect the construction of a program? Can software flexibility, a major reason for distributed computing, be maintained? Can established software engineering criteria for program quality be satisfied? Can programs written in this way satisfy real-time constraints?

These questions can only be answered definitively in a practical, real-world context. Unfortunately, those kinds of answers require tremendous amounts of time and money, so, given finite time and resources, the solution is to test the theory incrementally. Instead of collecting a quantity of requirements documents, implementing them to the letter, and then evaluating the results, we choose to concentrate on one application only. The basic idea is to abstract from the

† Ada is a trademark of the U.S. Department of Defense

application a requirements document that is specific enough to be realistic and general enough to be used in a variety of contexts. Then various program structures on various target architectures could be easily constructed and rapidly evaluated for practical viability.

We believe this document could be useful to other researchers who face similar difficulties with practical evaluation of software engineering constructs. Admittedly, the credibility of this still-theoretical kind of analysis is not equivalent to that of an evaluation of a fully detailed, real-world system. In particular, we expect to unearth no incontrovertible evidence as to the capacity of a system to meet real-time deadlines. We do expect our analysis to yield useful insights available only from the study of a plausible software structure, and we anticipate a sizable savings in effort over the more credible all-out implementation effort.

The application is an aircraft navigation and control system capable of automatically performing all flight tasks from take-off through landing. The remainder of this document focuses on particulars of the application.

2. SYSTEM INPUTS

All inputs to the system fall in one of six general categories:

Onboard navigation sensors provide the software with situational information. These devices measure such quantities as wind speed, position, velocity, acceleration, roll, pitch, and yaw. Due to their unreliability, all navigation sensors are duplicated or triplicated.

Ground-based navigation aids supplement the onboard navigation sensors. During normal flight, angle and distance data from radio beacons at known locations allow the software to compute aircraft position accurately. The computed position then serves as a correction to the filters which integrate acceleration into velocity and position. Different, more accurate,

navigation aids are used in the vicinities of airports. Most airports provide a navigation aid called the Instrument Landing System (ILS). ILS is accurate enough to permit fully automatic landings; however, curved approaches cannot be supported because of the narrow range of the transmitters. Certain airports provide the more advanced Microwave Landing System (MLS). Because of the sixty degree horizontal range, curved approach landings can be fully automated.

Analog control devices, linked to the computing system by position sensors, allow the pilot to control the aircraft directly. A joystick, known as the brolly handle, permits the pilot to control, via the ailerons and elevators, the attitude of the aircraft. A foot pedal in the cockpit affords the pilot direct control of the rudder. An accelerator pedal controls the throttle.

The **Flight Control Panel** (FCP) allows the pilot to select the manner in which the flight of the aircraft will be controlled. Some modes selectable via this panel supplement the analog control devices by causing the software to maintain current attitude, flight path, or track angle. This panel also allows the pilot to bypass the analog control devices completely. Knobs on the panel can be used to specify digitally the desired speed, altitude, flight path angle, and track angle. A final set of modes which direct the software to control the aircraft automatically by comparing position to the flight plan can also be selected on the flight control panel.

The **Navigation Control Display Unit** (NCDU) is the medium by which the pilot can inspect and modify information pertinent to the flight plan. He can use it to request general information about airports, runways, and navigational aids and to input or to modify the current flight plan.

The two **display control panels** allow the pilot to select display formats for the Primary

Flight Director (PFD) and the Navigational Display (ND).

2.1. SYSTEM OUTPUTS

All outputs from the system fall in one of five general categories:

Effector outputs control flight of the aircraft. These outputs include commands to the ailerons, delta elevators, rudder, and throttle. The commands take the form of positions to which the various control surfaces and the throttle should move.

The **FCP indicator lights** inform the pilot of the level of automatic flight assistance currently being offered. Digital readouts on this panel tell him the speed, altitude, flight path angle, and track angle. These values are either current or desired as indicated by the lighting configuration of the panel.

Character outputs to the **NCDU screen** provide appropriate feedback to the keyboard requests of the pilot. Information about airports, runways, the current flight plan, navigation sensors, and navigational aids can be displayed on the NCDU screen.

The **PFD and ND screens** are the devices by which the pilot can monitor the situation of the aircraft. The PFD screen displays the orientation of the aircraft in space. The ND screen displays the progress of the aircraft as compared to the flight plan.

The **navigation radio tuner** adjusts the radio to receive the frequencies transmitted by the ground-based beacons which provide correction data to the navigation function.

2.2. FUNCTIONAL UNITS

The application can be divided into three major functional units. Navigation computes the situation of the aircraft. The concept of situation includes such measurements as position, velocity, acceleration, roll, pitch, and yaw. Various filters within the navigation unit compute these quantities from flight sensor readings and ground aid transmissions.

Several sensor systems with overlapping functions are used by the navigation unit. Microwave Landing System (MLS) transmitters, located on runways in a few airports around the U.S.A., provide the most accurate measurements of latitude, longitude, and position. When the aircraft is in the vicinity of these transmitters, integration of acceleration measurements from the Inertial Navigation System (INS) gyros on board the plane use the MLS-derived position as a filter correction. The secondary ground-based landing aid is the Instrument Landing System (ILS), which supplies the correction term to the filter near runways which are not equipped with MLS. When the aircraft is not landing, data from radio beacons at known locations on the ground allow computation of the correction term. Other sensors measure true air speed, magnetic heading, magnetic variation, and barometric altitude, and these sensors comprise a correction and backup system to the other navigation sensor systems.

The second major function of the application is flight control, which can be divided into two parts, automatic guidance and the manual control laws. Both kinds of flight control generate positions for the aircraft control surfaces and throttle. They differ primarily in the level of automation provided.

Guidance compares the aircraft situation to the flight plan and generates control surface and throttle positions which move the vehicle into accord with the flight plan. In addition to the current flight plan and the navigation results, guidance uses as inputs the computed engine

pressure ratio limits and the current control surface positions. These supplementary data impose limits on the actions which can be directed by the guidance routines. For example, a guidance computation which would cause the vehicle to accelerate to the point of exceeding the engine pressure ratio limits will be limited to a safe level. Similarly, drastic changes in the control surfaces are limited to avoid a rough ride and possible physical damage to the aircraft.

Horizontal guidance manages the direction (in the common sense) of the aircraft by controlling the rudder and ailerons. Waypoints, or locations over which the aircraft should pass, are input via the cockpit, and horizontal guidance steers the vehicle from one waypoint to the next in straight-line fashion. Vertical guidance manages the altitude of the aircraft by controlling the delta elevators. When vertical guidance is engaged, each waypoint will have an altitude associated with it in addition to the requisite latitude and longitude data. An aircraft using vertical guidance will travel between each waypoint with a constant rate of altitudinal change. Time guidance manages the acceleration of the plane by controlling the throttle. Arrival times associated with each waypoint establish, between waypoints, a constant velocity which situates the vehicle over each waypoint at the specified time.

Horizontal, vertical, and time guidance are individually selectable on the flight control panel; however, vertical guidance is disabled until horizontal guidance is engaged, and time guidance cannot engage until vertical guidance has done so. Also necessary for engagement of a particular level of guidance is specification of a flight plan specific enough for that level. For example, time guidance cannot operate until arrival times are associated with each waypoint in the path.

The second kind of flight control, the manual control laws, performs the same basic function as does automatic guidance; however, the inputs and therefore the actual computations differ dramatically. Instead of using path and navigation data, the manual control laws translate

analog pilot directives into positions for the control surfaces and the throttle.

Three control laws, abstracted from the analog computers which used to operate airplanes, comprise the heart of the unit. The lateral control law, analogous to horizontal guidance, computes the aileron and rudder commands. The elevator control law, analogous to vertical guidance, computes the delta elevator command. The throttle control law, analogous to time guidance, computes the throttle command.

Different levels and kinds of automatic assistance are available to the pilot in this manual control mode. Also, for every flight mode the software prevents the pilot from causing the plane to stall or incur physical damage. For example, if someone were to fall on the brolly handle in such a way as to push it to some extreme, the software would obey the directive only to a point short of disaster.

The simplest mode is called attitude control wheel steering (ACWS). ACWS maintains the existing pitch and roll, the attitude, of the aircraft whenever the pilot returns the brolly handle to the detent position. When the brolly handle is not at detent, the aircraft in ACWS mode simply translates the position of the joystick into appropriate behavior of the vehicle control surfaces.

Velocity control wheel steering is similar to ACWS but offers somewhat more assistance. Instead of maintaining the attitude of the vehicle when the pilot returns the brolly handle to detent, VCWS holds steady the velocity vector of the aircraft. Unlike ACWS, then, VCWS controls the throttle as well as the control surfaces.

The third and final function of the application is pilot information. This function provides formatted data to the two horizontal situation indicators and the attitude director indicator for display to the pilot. The three control panels associated with the indicators indicate the formats

and contents of the display output data. Navigation and guidance results are the inputs to the display unit.

3. SYSTEM DATA FLOW

The following diagrams depict the data flow of the example application. Dashed boxes indicate functional units which are more fully described at a higher level of abstraction. Dotted boxes indicate functional units or groups of units which are more fully described at a lower level of abstraction.

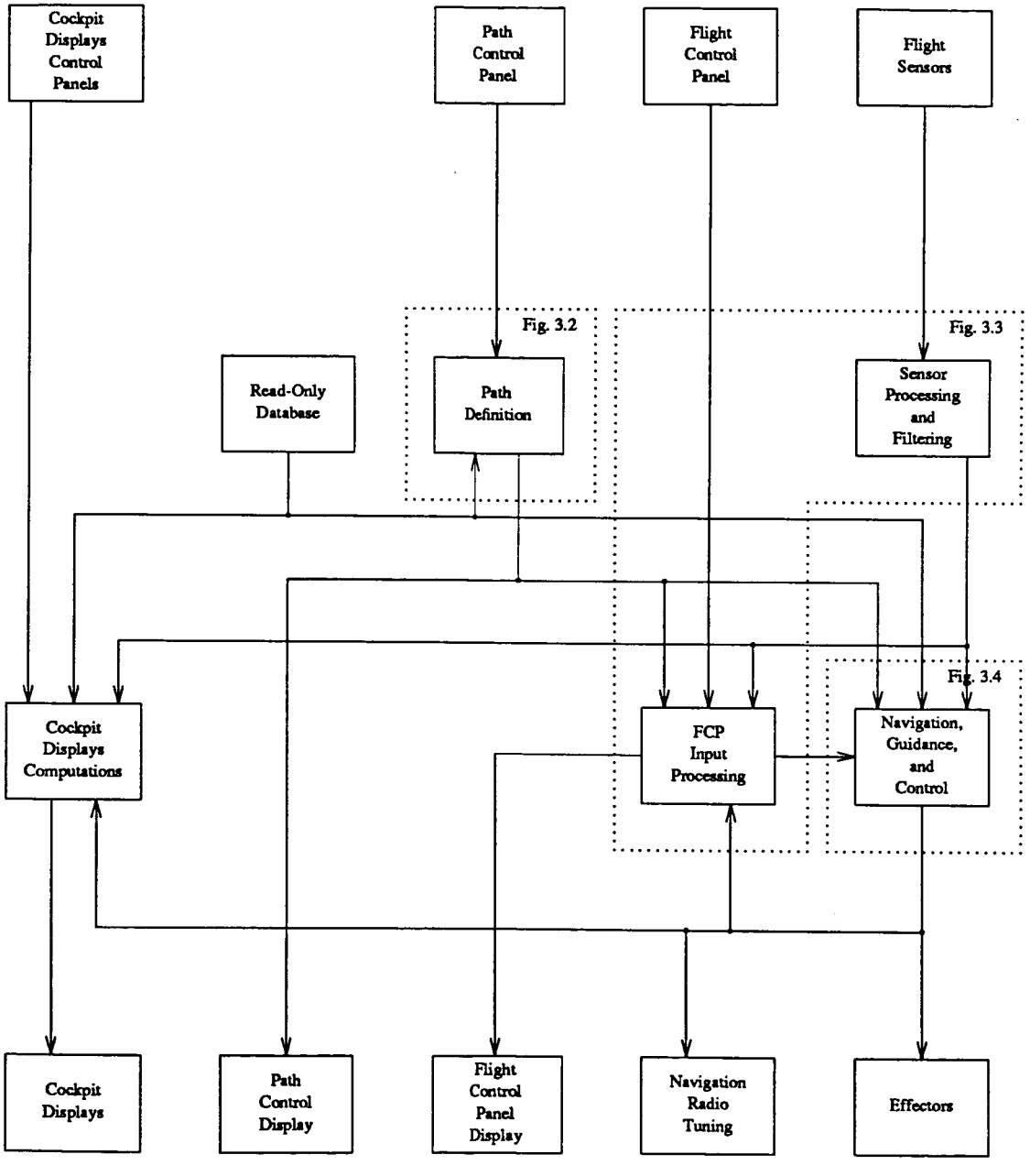


Fig. 3.1. System Data Flow

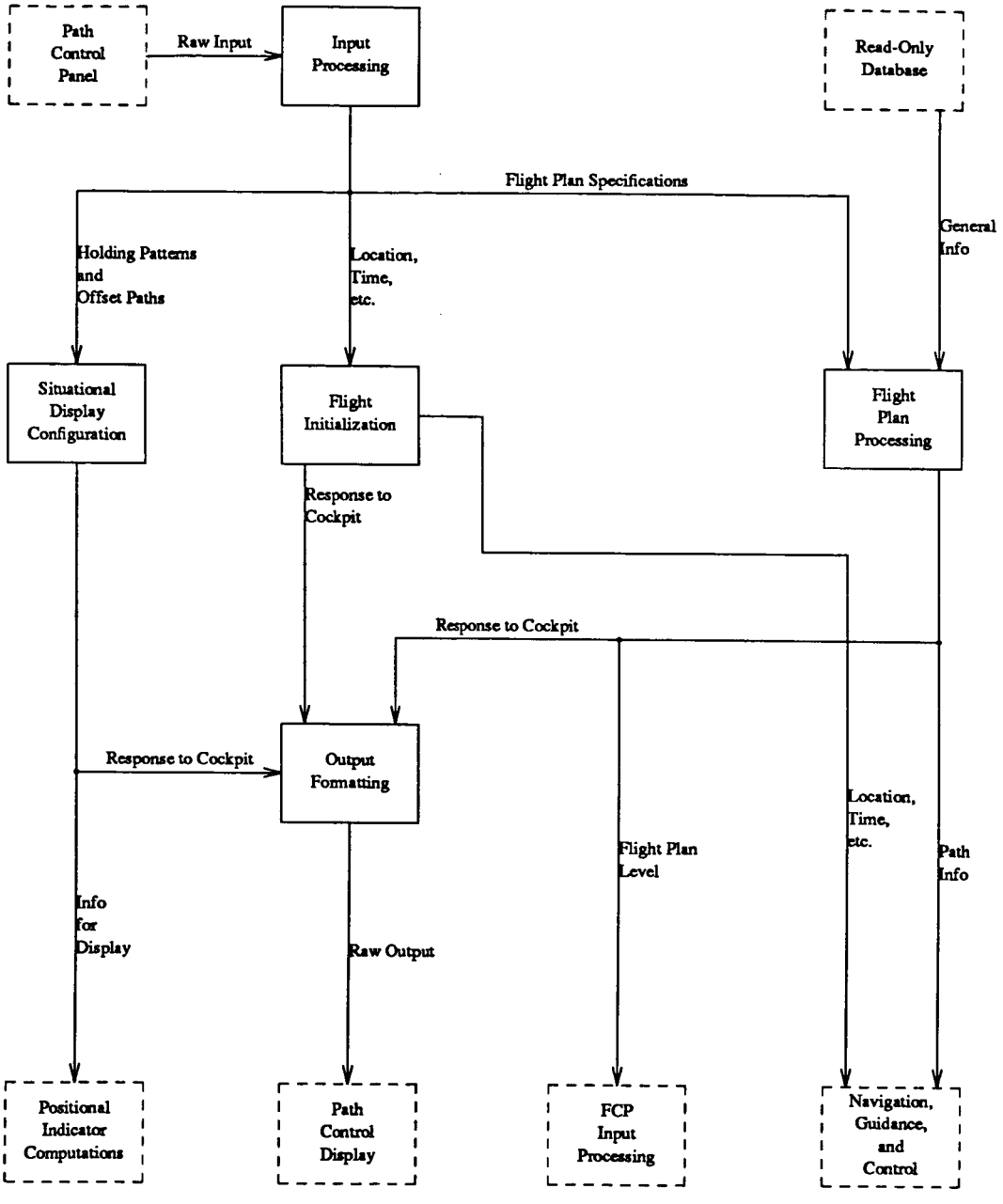


Fig. 3.2 Path Definition Data Flow

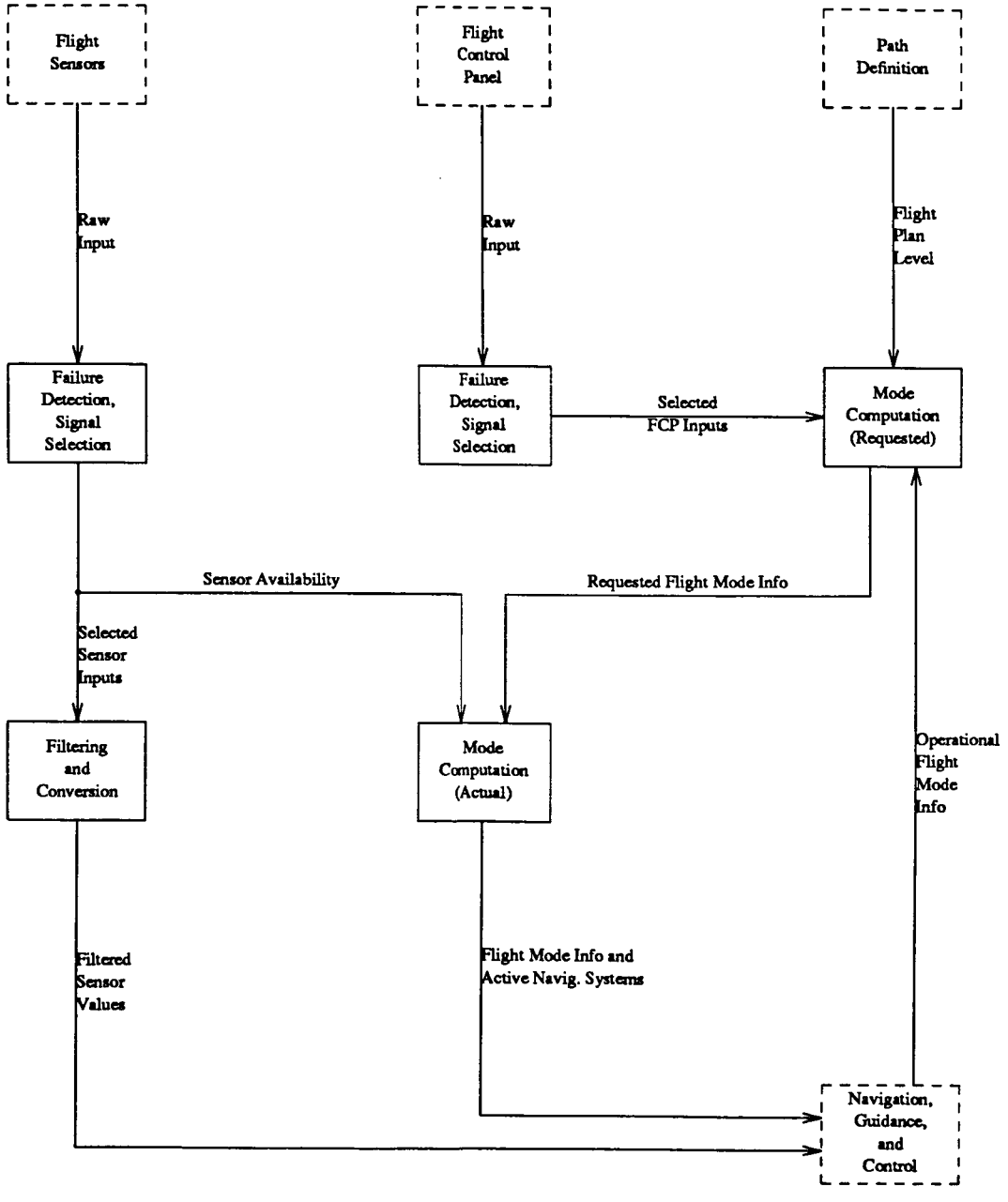


Fig. 3.3 Flight Controls and Flight Sensors Input Processing Data Flow

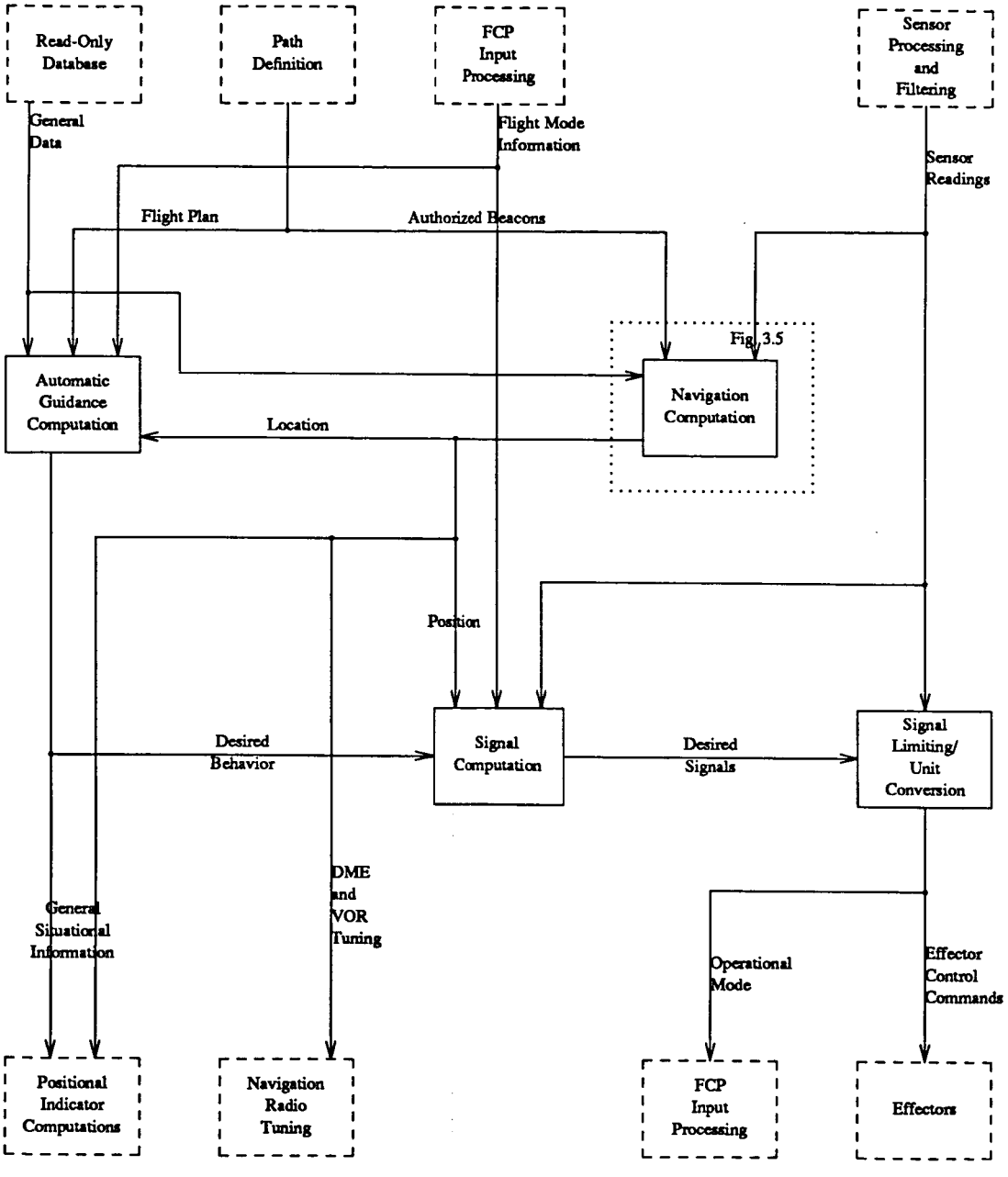


Fig. 3.4 Navigation, Guidance, and Control Data Flow

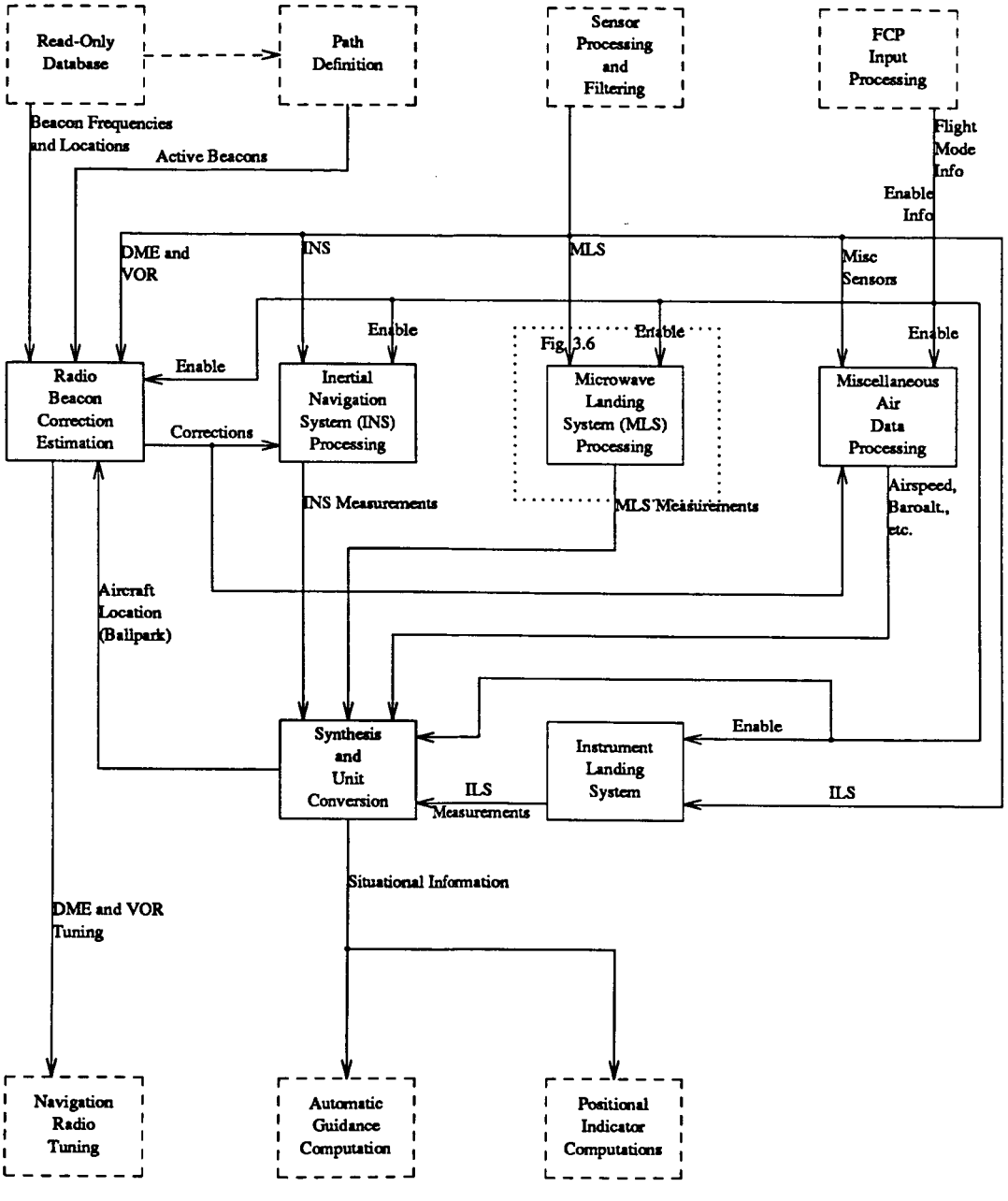


Fig. 3.5 Navigation Data Flow

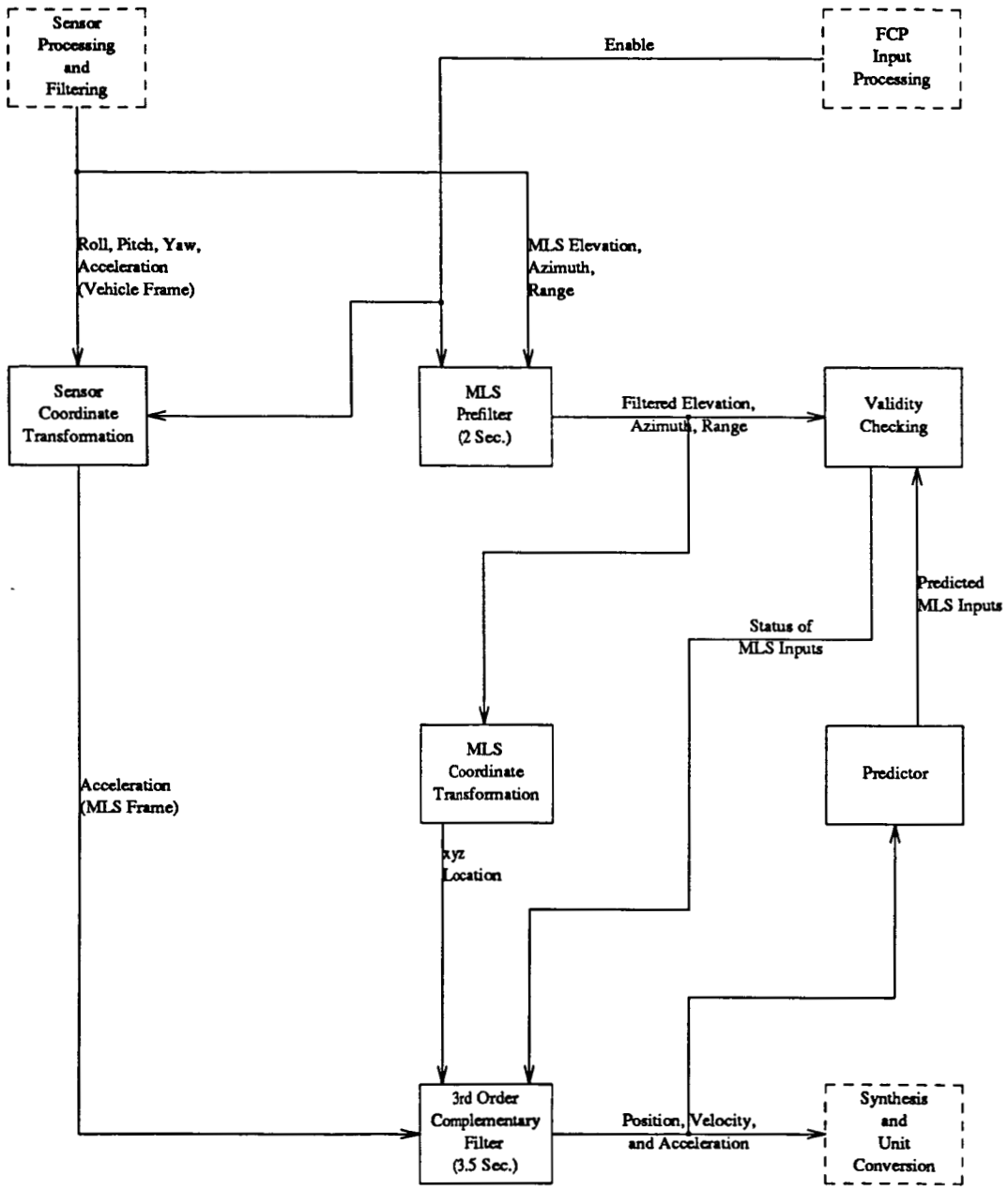


Fig. 3.6 Microwave Landing System (MLS) Data Flow

4. REAL-TIME CONSIDERATIONS

Each function operates either in the foreground or in the background. The clock granularity is ten milliseconds, and typical foreground tasks execute once every fifty milliseconds. The foreground functions can be grouped according to the approximate relative lengths of time they require to execute (if the situation demands their execution). Functions in group A execute in one relative time unit. Those in groups B and C require two and three relative time units, respectively. For example, execution of a function from group A is one-third as costly as is execution of a function from group C. The following functions belong to group A:

Horizontal guidance
Vertical guidance
Time guidance
ILS processing
INS processing
Navigational aid processing

The following functions belong to group B:

MLS processing
Lateral control law
Vertical control law
Throttle control law
Sensor error detection and signal selection

The following functions belong to group C:

ND processing
PFD processing

5. DESCRIPTION OF CSC IMPLEMENTATION

Computer Sciences Corporation developed and maintains the ATOPS software for NASA. They implemented the system on two computers. One computer, called the Displays Computer (DC), is responsible largely for such non-crucial tasks as the computation and formatting of information for the cockpit displays. The other computer, called the Flight Management and Control Computer (FM/FC), handles the more critical aspects of the duties of the computing system. It is with the software on this computer that we are concerned here.

5.1. Timing

Time in this implementation is divided into *minor frames*, five of which comprise a *major frame*. Minor frames are of ten milliseconds duration, and major frames last for fifty milliseconds. All nodes have access to the same clock, so major and minor frames begin at the same real times at all nodes. See figure B.1 for the expected flow of control within a typical major frame.

5.2. Fast

The fast system functions are executed every ten milliseconds. Computations in this system proceed at a fifty-millisecond rate; however, certain calculations are so sensitive to time that as inputs they must have sensor readings which are current to within ten milliseconds. Also, the effectors associated with these calculations must receive the results within ten milliseconds of their computation. One duty of the fast loop, then, is to handle the time-sensitive inputs and outputs.

The rest of the inputs and outputs are handled once every major frame. They are received and sent en masse in order to reduce the overhead associated with a piecemeal approach. The fast

Minor Cycle #1 (0-9 ms)	Send all outputs. Receive all inputs. Begin execution of medium speed functions.
Minor Cycle #2 (10-19 ms)	Send 10 ms outputs. Receive 10 ms inputs. Continue execution of medium speed functions.
Minor Cycle #3 (20-29 ms)	Send 10 ms outputs. Receive 10 ms inputs. Finish execution of medium speed functions. Continue execution of slow speed functions.
Minor Cycle #4 (30-39 ms)	Send 10 ms outputs. Receive 10 ms inputs. Satisfy inter-node communication requirements. Continue execution of slow speed functions.
Minor Cycle #5 (40-49 ms)	Send 10 ms outputs. Receive 10 ms inputs. Continue execution of slow speed functions.

Fig. 5.1 Major Cycle Timing Example

loop performs this function once out of each five iterations.

5.3. Medium

The following list details the operation of the medium loop. The functions are listed in the order of execution during each iteration:

(1) **Input.** Check required discrete and analog inputs for errors. Format inputs for use by other modules.

(2) **Flight control panel.** Calculate desired automatic flight modes. Prepare lighting configuration and window values for output to flight control panel (less the four bottom-left buttons).

(3) **Navigation control display unit.** Acquire any NCDU inputs. Configure slow NCDU process for general activities required by inputs. Compute selected outputs to NCDU. Send all NCDU outputs.

(4) **MLS.** Compute aircraft position, velocity, and acceleration vectors from the received MLS signals if microwave landing system mode is enabled.

(5) **Flight control mode.** Calculate the aircraft flight mode, which configures the flight control laws. Based upon the possible inclusion of an airport in the flight plan and the validity status of the microwave landing system, assign alternate sources for some navigation and flight control variables. Prepare lighting configuration for output to the four bottom-left buttons on the flight control panel.

(6) **Navigation.** Compute aircraft heading, position, velocities, and accelerations based on any of several sources. Compute a number of useful quantities from the fundamental results. If MLS computations are valid, derive all outputs except true heading from MLS results. If MLS computations are not valid, and INS sensors are functional, use the INS results as corrected by the radio navigation computations performed by the navigation slow loop. If the MLS and INS are not operational, calculate the navigation solutions from true air speed, magnetic heading, and magnetic variation. For both INS and air data solutions, derive the vertical position, velocity, and acceleration results from the barometric altitude and INS vertical acceleration measurements.

(7) **Horizontal guidance.** Execute only when 2D guidance is enabled. Compare calculated aircraft latitude and longitude to desired path. Generate lateral steering signal for use by lateral control laws.

(8) **Vertical guidance.** Execute only when 3D guidance is enabled. Compare calculated aircraft altitude to desired path. Generate vertical steering signal for use by vertical control law.

(9) **Time guidance.** Execute only when 4D guidance is enabled. Compare calculated aircraft position to desired position at current time. Generate acceleration command for use by throttle position software. Generate display and NCDU inputs.

(10) **Automatic flight mode.** Based upon guidance computations and the desired automatic flight mode computed by the flight control panel software, calculate the actual flight mode. Provide the lateral and vertical control laws and the throttle position software with inputs based upon the actual flight mode.

(11) **Lateral control laws.** Compute the aileron and rudder commands from control inputs provided by the automatic flight mode software and from control inputs selected according to the actual flight mode.

(12) **Vertical control law.** Compute the delta elevator command and the stabilizer trim discretes from control inputs provided by the automatic flight mode software and from control inputs selected according to the actual flight mode.

(13) **Throttle position.** Compute the throttle position from the control inputs provided by the automatic flight mode software and from control inputs selected according to the actual flight mode.

(14) **Sensor utilization.** Determine which of the inputs subject to error checking are in use and therefore need to be checked during the next cycle.

(15) **Output.** Format outputs to effectors and cockpit.

5.4. Slow

The following list details the functions of the FM/FC slow loop in the order in which they are executed iteratively:

(1) **Navigation Control Display Unit.** Parse NCDU input. Perform functions requested by NCDU input. Format NCDU output pages. Send resulting pages to NCDU fast loop executive for eventual output. Define aircraft flight path and distribute it to the functions which use it.

(2) **Navigation.** Tune navigation radio receivers. Compute position and velocity corrections to fast loop navigation computations. Provide display software with navigational mode information.

(3) **Earth radius.** Compute radii of curvature in the east-west and north-south planes. Set magnetic variation to INS measurement of it if INS navigation is enabled.

(4) **Wind velocity.** Calculate wind speed and direction. Make the results available to the display software.

(5) **Engine pressure ratio limits.** Calculate the maximum engine pressure ratios for the climbing, cruising, and continuous thrust situations. Make the three calculations available to the throttle position software.

5.5. Top-Level Code

The following Ada code provides a detailed description of the overall organization of the two-computer CSC implementation:

```
procedure AutomaticFlightControl is

package FlightManagementAndControl is
   type ToDisplays is
   record
      -- components of variant record to be sent to displays computer
   end record;

   task FastFMandC is
      for FastFMandC use at (Cpu1, Cpu1Mem1, UndefinedAddress);
      pragma priority (75);
      entry ClockTick;
      for ClockTick use at (Cpu1, Cpu1Mem1, ...);
   end FastFMandC;

   task MediumFMandC is
      for MediumFMandC use at (Cpu1, Cpu1Mem1, UndefinedAddress);
      distribute_to (Cpu1);
```

```
      pragma priority (50);
      entry Iterate;
   end MediumFMandC;

   task SlowFMandC is
      for SlowFMandC use at (Cpu1, Cpu1Mem1, UndefinedAddress);
      pragma priority (25);
      entry Start;
   end SlowFMandC;
end FlightManagementAndControl;


package body FlightManagementAndControl is
   task body FastFMandC is
   begin
      frameCounter := 0;
      loop -- forever
         accept ClockTick;
         frameCounter := (frameCounter + 1) mod 5;
         if frameCounter = 0 then
            SendAllOutputs;
            ReceiveAllInputs;
            MediumFMandC.Iterate;
         else
            Send10MsOutputs;
            if frameCounter = 4 then
               FastDisplay.FMandC (dataForDisplaysComputer);
            end if;
            Receive10MsInputs;
         end if;
      end loop;
   end FastFMandC;


   task body MediumFMandC is
   begin
      SlowFMandC.Start;
      loop -- forever
         accept Iterate;
         FormatSensorInputs;
         ModeSelectPanelLogic;
         NcduFast;
         Mls;
         FlightControlMode;
         NavigationFast;
         HorizontalGuidance;
         VerticalGuidance;
         TimeGuidance;
         AutomaticFlightMode;
         LateralControlLaws;
         VerticalControlLaw;
```

```
            ThrottleControlLaw;
            SensorUtilization;
         end loop;
      end MediumFMandC;


      task body SlowFMandC is
      begin
         accept Start;
         loop -- forever
            NcduSlow;
            NavigationSlow;
            EarthRadius;
            WindSpeedandDirection;
            EnginePressureRatioLimits;
         end loop;
      end SlowFMandC;
   end FlightManagementAndControl;


   package PilotDisplays is
      task FastDisplay is
         for FastDisplay use at (Cpu2, Cpu2Mem1, UndefinedAddress);
         pragma priority (75);
         entry ClockTick;
         for ClockTick use at (Cpu2, Cpu2Mem1, ...);
      end FastDisplay;

      task MediumDisplay is
         for MediumDisplay use at (Cpu2, Cpu2Mem1, UndefinedAddress);
         pragma priority (50);
         entry Iterate;
      end MediumDisplay;

      task SlowDisplay is
         for SlowDisplay use at (Cpu2, Cpu2Mem1, UndefinedAddress);
         pragma priority (25);
         entry Start;
      end SlowDisplay;
   end PilotDisplays;

   package body PilotDisplays is
      task body FastSpeed is
      begin
         frameCounter := 0;
         loop -- forever
            accept ClockTick;
            frameCounter := (frameCounter + 1) mod 5;
            if frameCounter = 0 then
               ReceiveAllInputs;
               MediumDisplay.Iterate;
```

```
            else
               Receive10MsInputs;
               if frameCounter = 2 then
                  SendSensorSelections;
               elsif frameCounter = 3 then
                  accept FMandC (data: in BigArray) do
                     dataFromFMandCComputer := data;
                  end FMandC;
               elsif frameCounter = 4 then
                  SendDisplayOutputs;
               end if;
            end if;
         end loop;
      end FastDisplay;


      task body MediumDisplay is
      begin
         SlowDisplay.Start;
         loop
            accept Iterate;
            SelectSignalsAndDetectFailures;
            FormatPanelInputs;
            FormatSensorInputs;
            ComputePfdValues;
            FormatPfdOutputs;
            NavDisplayFast;
            PrepareAllOutputs;
         end loop;
      end MediumDisplay;


      task body SlowDisplay is
      begin
         accept Start;
         loop
            NavDisplayBackground;
         end loop;
      end SlowDisplay;
   end PilotDisplays;
end AutomaticFlightControl;
```

APPENDIX 3

ANALYSIS OF Ada FOR A CRUCIAL DISTRIBUTED APPLICATION

ANALYSIS OF Ada† FOR A CRUCIAL DISTRIBUTED APPLICATION‡

John C. Knight Marc E. Rouleau

Department of Computer Science
University of Virginia
Charlottesville, Virginia, 22903

SUMMARY

Ada was designed for the programming of embedded systems, and has many characteristics intended to promote the development of reliable software. Many embedded systems are distributed, and an important characteristic of such systems is the ability to continue to provide adequate though perhaps degraded service after loss of a processing node.
We are concerned with the issues that arise when critical distributed systems are programmed in Ada.
We have shown previously that numerous aspects of Ada make its use on distributed systems problematic if processor failures have to be tolerated. The issues are not raised from efforts to implement the language but from the lack of semantics defining the state of an Ada program when a processor is lost.
We have suggested appropriate semantic enhancements to Ada and have described the support system required to implement these semantics.
We are evaluating and refining this approach by applying it to a large-scale aerospace system.
The application is the resident software for a modified Boeing 737 transport equipped with a fully automated, digital, control system.
The framework for a fault-tolerant version of the complete system has been constructed, and certain critical functions have been programmed in their entirety.
In this paper we present a summary of the application, details of its implementation in Ada, and a preliminary evaluation of the utility of Ada in this context.

---

† Ada is a trademark of the U.S. Department of Defense

‡ This work was supported in part by NASA grant NAG1-260

INTRODUCTION

A distributed system is only as reliable as its weakest node. For example, if each of three computers in a system will operate correctly 99 percent of the time, the complete system will function, on average, only approximately 97 percent of the time. If a node has failed, however, a distributed system has the potential to continue providing service since some hardware facilities remain. Success in this endeavor will make these systems *more* reliable than single-processor architectures. More importantly, distributed systems can, if necessary, provide degraded service following the failure thereby allowing either reduced service or a timely, controlled shutdown. In real-time control applications requiring very high reliability, such as are found in the defense and aerospace areas, this can be extremely important.

Ada[1,2] has been designed for such critical, real-time applications and for the development of programs distributed over multiple computers. Unfortunately, it has been shown to be deficient in this area[3]. In fact, the Ada definition ignores the problem completely and implies that the Ada machine does not fail. With this in mind, some researchers have proposed a *transparent*[4,5] approach to recovery in which an application program is unaware of the fault-tolerant capabilities of the system. Loss of a processing node would cause automatic reconfiguration of the system, and the reconfiguration would be invisible to the program. Operation of the system before and after node failure would be identical. The burdens of recovery and of the preparation needed for that recovery are placed upon the execution-time environment.

Knight and Urquhart[3] have suggested a method by which system designers can specify explicitly the service to be offered following node failure. This allows for the specification of degraded, or *safe*[6], service in systems which, due to lack of computing power, cannot provide full functionality after a hardware failure. In addition, with this approach designers control the normal operation overhead required to prepare the software for an arbitrary hardware fault. This

overhead consists primarily of transmissions of critical data items required during reconfiguration to bring the system to a consistent state. This method is termed the *non-transparent* approach to tolerating the loss of a processor in a distributed system.

Previous research has shown that the non-transparent method is *theoretically* feasible. A distributed testbed containing an execution-time system which provides the necessary facilities for non-transparent recovery has been constructed[7]. A simple distributed Ada program, written according to certain guidelines, can be run on the testbed, and arbitrary failure of one of the nodes will initiate reconfiguration and the return to "acceptable" operation (lacking one computer).

The goal of the research described here is to evaluate the *practical* value of non-transparent recovery as proposed by Knight and Urquhart by analyzing their feasibility in the context of one realistic application. The application is NASA's Advanced Transport OPerating System (ATOPS), an aircraft computing system capable of performing all flight tasks from take-off through landing. While the resultant data point will provide, in and of itself, little persuasive evidence for or against the proposed non-transparent method, the associated analysis should provide useful insight into the issues relevant to hardware-fault-tolerant Ada distributed systems.

This paper is organized as follows. In the next section we summarize the issues surrounding the use of Ada on fault-tolerant distributed systems. The application being examined in this experiment is then described, and the design of the experiment, including the criteria used in evaluation, is then presented. The preliminary analysis of the experiment is then discussed and finally a summary and conclusions are presented.

Ada SEMANTICS AND NON-TRANSPARENT CONTINUATION

In this section we summarize the difficulties with Ada semantics and the approach to non-

transparent continuation suggested by Knight and Urquhart. For more details see reference 3.

Deficiencies In Ada Semantics

There are two categories of semantic deficiency in Ada. First, the definition of Ada lacks any clear definition of the meaning of a distributed Ada program, i.e., there are no *distribution semantics*. Specification of the basic units of distribution, whether they be individual statements, tasks, packages, or anything else, is omitted as is the detailed meaning of such a program. This is surprising since representation clauses are available to control many other implementation areas in great detail.

The lack of distribution semantics is not a simple problem to solve. It is not sufficient merely to define what can be distributed. It is also necessary to define exactly what distribution of an object will mean. For example, if a task can be distributed, it is essential that the meaning of the distribution include details of the location of the associated code and data. A syntax to control distribution is also required, of course.

The second semantic deficiency is the lack of definition of the meaning of a program following a hardware failure. An Ada program operating on a distributed target is obviously going to be damaged by the loss of a node from the system. The extent of the damage must be precisely defined since it goes beyond the loss of the software on the failed node and affects the software on nodes that survive failure. For example, if two tasks are engaged in a rendezvous when a failure occurs, the caller would be permanently suspended if the server was lost since the rendezvous would never end and the caller could not distinguish this situation from slow service by the server.

Another area where damage to tasks surviving failure can occur is the possibility of loss of context. If a nested task survives failure but the surrounding task is lost, the nested task may lose

part of its context (the local variables for the surrounding task) and cannot be accessed since no remaining undamaged tasks can know its name.

The *failure semantics* of a programming language define precisely the state of the software that remains following the loss of a processor at an arbitrary point. Failure semantics for Ada must be defined if the software that remains is to be in a form that can be predicted accurately during design. This is necessary if some form of service is to be provided reliably after a failure.

Non-Transparent Continuation

Distribution and failure semantics for Ada that involve no syntactic changes other than the definition of a pragma have been defined for Ada[3]. The major component of distribution in the approach is the task. A program is structured as a set of tasks with a main program that consists solely of a null statement. The tasks nested within the main program can be distributed as required using a pragma. Tasks nested at lower levels are allocated to the processor of the surrounding task(s).

Failure semantics are defined to be equivalent to abort semantics. Where a task is lost through hardware failure, it is assumed that it was in fact aborted and the effect on the remaining program is therefore well defined. It is necessary to supplement the usual Ada execution-time support system in order to provide these semantics. Continuing with the example of the broken rendezvous used above, in that case it is assumed that the lost task was aborted and the support system will be required to raise a tasking error exception in the calling task.

Each processor has a *reconfiguration* task associated with it and this task receives a call to a predefined entry when a processor failure occurs. The support system is required to monitor processor health and signal the loss of of a processor to those that remain by making this entry call when necessary. The reconfiguration task executes code that takes care of the needs of the

processor on which it resides. Alternate tasks for those that have been lost reside on remaining processors and are activated by the reconfiguration task using entry calls.

A necessity in any form of continuation is the availability of data that survives the failure. Many components of an application update local data that must be available on each real-time cycle. To recover such a function, a consistent copy of the local data must be available on the processor that executes the alternate software. In a distributed system the reliable distribution of such data can be achieved with a two-phase commit protocol[8,9]. In the proposed non-transparent continuation, the data to be made consistent is determined by the programmer and the times when the copies are updated are the programmer's responsibility. Data distribution is achieved by rendezvous with a *data consistency* task that implements the two-phase commit protocol.

All inter-task communication takes place using the features provided by Ada whether this is inter- or intra-machine communication. This provides uniformity for the programmer and allows compile-time checking. Following failure, however, services will be resumed by alternate tasks to replace those lost by failure and these tasks will have different names. This means that all inter-machine communication will have to be programmed with explicit selection of the entry to be used. This has a substantial effect on program structure and hence on performance.

EXPERIMENT DESIGN

Our approach in this research is to attempt to construct software in Ada to meet the specifications of the ATOPS system and to incorporate tolerance to hardware failures also. Analysis is performed during and after construction to determine the success or otherwise of the non-transparent approach.

No single conclusion about the utility of non-transparent continuation can be drawn from this research since what constitutes a serious limitation in one context might not be in another.

There are several distinct criteria that must be used in the evaluation; each criteria affects the overall determination of the success of the method in any given application. These criteria fall into three broad categories - development-time issues, execution-time issues, and the effect of different target architectures. In this section we discuss the criteria we are using and the reasons for their use.

Development-Time Issues

Non-transparent continuation allows the programmer to determine the details of recovery following failure. The reconfiguration software and the software for alternate services must be prepared during development but we do not consider this software as overhead since it constitutes the implementation of a substantially more useful application than an implementation with no recovery. However, its volume can be measured and memory space must be available for it.

It is very desirable to delay decisions about binding functions to processors as late as possible in the development process. A major reason for using a single program on a distributed target is flexibility. A function that is written as a distributable entity, say a task, can be moved to a different processor quite simply if the task is merely part of a single program. All that is required, in principle, is a change to the distribution directive, recompilation, and relinking. Non-transparent continuation reduces this flexibility since, if the function being moved has to survive processor failure, alternate software must be provided. Thus, moving a function requires not only the possible movement of the alternate software, but also the revision of the reconfiguration tasks, changes to the use of the data distribution tasks, and possibly changes to the other software. For example, some communication may have to be modified to be prepared for redirection, and other communication may have to be modified to remove the ability for redirection.

Another area of flexibility provided by distributed systems is the ease of incremental change in computing performance. If, during development, it is discovered that the estimate of required performance is incorrect, processors can be added or deleted as needed. Adding or removing a processor to or from a system incorporating non-transparent fault tolerance amounts to requiring the movement of several functions at once from one processor to another. It is important to ensure that non-transparent continuation does not reduce the flexibility of distributed systems to the point where useful flexibility is lost.

A final concern at development time, is the possibility that the inclusion of non-transparent continuation may so distort the desired form for a program that properties of programs in the software engineering sense, such as good modularity, information hiding, etc., might be lost. These are difficult properties to quantify and are, to a large extent, subjective, but any indication that these properties must be sacrificed for recovery would be serious.

Execution-Time Issues

At execution time, the major concern is overhead. The resources used by non-transparent continuation must not reduce overall performance to the point where real-time deadlines cannot be met. Areas where overhead will be incurred are:

(1) communications bus traffic,

(2) processor overhead used in data distribution,

(3) processor overhead used in the implementation of failure semantics,

(4) processor overhead used by the application to determine select between primary and alternate software, and

(5) memory space required to support alternate software.

An issue related to overhead is response time. It must possible for recovery to take place fast enough following the loss of a processor to ensure that the equipment being controlled does not suffer from a lack of service. The reconfiguration task must be started, must perform its services, and alternates must be started sufficiently quickly that real-time deadlines are not missed.

Target Architecture

All of the above criteria are affected in practice by the target architecture. The key elements of the architecture are:

(1) the number of processors provided,

(2) the types of the processors and the number of each of the different types,

(3) the relative performance levels of the processors, and

(4) the sizes of the supplied memories.

Clearly, if a system is operating with all resources fully used, it will be impossible to maintain all services following failure. In fact, if all available memory is used in providing the original services, no fault tolerance will be possible since there will be no memory space for alternate software or reconfiguration software.

In some systems, specialized processors are provided to allow processor organizations that are tailored to specific needs of the application. For example, an array processor or a fast-fourier-transform processor might be provided to enhance performance. Loss of this type of equipment has a different effect on a distributed system than loss of a general-purpose processor because it is unlikely that the remaining equipment will be able to provide service at the speed

required. Some compromise will be necessary. Similarly, it is unlikely that a specialized processor will be able to take over the services lost when a general-purpose processor is lost.

In this research we have selected a small number of architectures that we feel are typical and that can support the ATOPS application. We are investigating, separately, architectures involving two, three, and four homogeneous processors. We assume that there is sufficient memory on each to accommodate the necessary redundant software but that in each case almost all of the available computing resources are used by the original application. Thus each analysis is supplemented by an attempt to add a processor to each target being investigated.

APPLICATION DESCRIPTION

The application we are analyzing in this research is an experimental aircraft navigation and control system capable of performing automatically all flight tasks from take-off through landing. The production system operates on a modified Boeing 737 in a flight controls research program at NASA Langley Research Center. The operational software is written mostly in HAL/S[10]. In principle, failure of such a computing system could entail loss of control of the aircraft, so the application is a candidate for fault tolerance. In practice, the HAL/S system operates with a safety pilot and a complete set of backup controls that operate conventionally.

Our analysis is theoretical and is based on rewriting only parts of the system in Ada. The software we have written is not intended to be used operationally, and none could ever be used on the actual aircraft. Our intention is to use the operational system as a realistic example so as to ensure that the analysis we perform takes all the real-world problems into account. We have assumed certain required functions are critical even though for this flight control application they might not be. Such functions would be critical in other applications and the reason for making these assumptions is to make the analysis more likely to apply generally.

The remainder of this section contains a complete but superficial description of the operation of the aircraft computing system. More detail can be found in reference 11.

Inputs

All inputs to the system fall into one of three general categories. First, aircraft sensors provide the software with situational information. These devices measure such quantities as wind speed, vehicle acceleration, flap positions, roll, pitch, and yaw. Due to their unreliability, all sensors are duplicated or triplicated. A major element of the system's software is management of the redundant sensors to ensure that computations are only performed with good sensors.

Ground-based radio and microwave navigation aids make up the second category of input. During normal flight, angle and distance information from radio beacons at known locations allows the software to compute aircraft position accurately. The computed position then serves as a periodic correction in the integration of acceleration into velocity and position. In the vicinities of certain airports, faster and highly accurate position information is available from Microwave Landing System (MLS) transmitters. The concomitant improvement in the accuracy of the location information allows fully automated landings.

The third and final source of input is the cockpit. An enhanced joystick, known as a *brolly handle*, permits the pilot to fly the plane in a conventional fashion through the computer system. The flight control panel allows the pilot to select the desired level of automatic flight assistance. This panel also allows him to specify digitally the desired speed, altitude, flight path angle, and track angle. The navigation control display keyboard is the medium by which the pilot can manage the navigation functions of the system. He can use the keyboard to request general information about airports and navigational aids, and to enter or to modify the flight plan. Two other control panels allow the pilot to select display formats for the two horizontal situation

indicators. A final control panel controls the display format of the attitude director indicator.

Outputs

All outputs from the system fall in one of two major categories. The first category is the effector outputs that control flight of the aircraft. These outputs include commands to the ailerons, delta elevators, rudder, and throttle.

The second category of system output is feedback to the cockpit. The lighting configuration of the flight control panel informs the pilot of the level of automatic flight assistance currently being offered. Digital readouts on this panel tell him the speed, altitude, flight path angle, and track angle. These values are either current or desired as indicated by the lighting configuration of the panel. Character outputs to the navigation control display screen provide appropriate feedback to the keyboard requests of the pilot. Information sent to the two horizontal situation indicators allows the pilot to observe the progress of the aircraft as compared to the flight plan. Data output to the attitude director indicator permit the pilot to monitor the orientation of the aircraft in space.

Functional Units

The application can be divided into four major functional units. Navigation computes the *situation* of the aircraft. The concept of situation includes such measurements as position, velocity, acceleration, roll, pitch, and yaw. The navigation unit computes these quantities from flight sensor readings and ground aid transmissions.

Several sensor systems with overlapping functions are used by the navigation unit. Microwave Landing System (MLS) transmitters, where available, provide the most accurate measurements of latitude, longitude, and position. When the aircraft is in the vicinity of these transmitters, integration of acceleration measurements from the Inertial Navigation System (INS)

gyros on board the plane use the MLS-derived position as a correction. A second less flexible ground-based landing aid, the Instrument Landing System (ILS), supplies the correction information near runways which are not equipped with MLS.

When the aircraft is not landing, data from radio beacons at known locations on the ground allow computation of the correction term. Other sensors measure true air speed, magnetic heading, magnetic variation, and barometric altitude, and these sensors comprise a correction and backup system to the other navigation sensor systems.

The second major function of the application is guidance. Guidance compares the aircraft position to the flight plan and generates appropriate steering and acceleration signals for the flight control laws. Horizontal, vertical, and time guidance are separately selectable on the flight control panel; however, vertical guidance is disabled until horizontal guidance is engaged, and time guidance cannot engage until vertical guidance has done so. Also necessary for engagement of a particular level of guidance is specification of a flight plan specific enough for that level. For example, time guidance cannot operate until arrival times are associated with each waypoint in the path.

Flight control is the third functional unit. Three control laws comprise the heart of the unit. The lateral control law computes the aileron and rudder commands, the vertical control law computes the delta elevator command, and the throttle control law computes the throttle command. Each control law uses inputs selected according to the flight mode. If no automatic guidance mode is engaged, the primary inputs will be from the brolly handle in the cockpit. If horizontal guidance is engaged, the lateral control law will use the lateral steering signal from guidance. If vertical guidance is engaged, the vertical control law will use the vertical steering signal from guidance. If time guidance is engaged, the throttle control law will use the acceleration signal from guidance.

The fourth and final function of the application is display. The display function provides formatted data to the two horizontal situation indicators and the attitude director indicator for display to the pilot. The three control panels associated with the indicators indicate the formats and contents of the display output data. Navigation and guidance results are the inputs to the display unit.

Timing

Each of the functions described above operates at one of three speeds. Certain critical inputs and outputs are handled every ten milliseconds. Most of the computations are performed every fifty milliseconds. Finally, some computations, especially those driven by inputs from a human source, are performed essentially as background functions whenever the ten and fifty millisecond computations are complete.

PRELIMINARY ANALYSIS

In our preliminary analysis, we have concentrated on the target architecture that is presently being used by the operational ATOPS system. This is a dual processor configuration in which flight-management and flight-control (FM/FC) functions reside on one computer, and sensor management and display functions reside on the other. This original partitioning of functions was determined by the desire for co-location of similar functions. In practice, it allows the existing operational system to meet its real-time deadlines. For identification, we refer to the two computers as the "FM/FC" and "Displays" computers respectively.

We have maintained this general partitioning and added recovery to it. This approach has the virtue of determining the feasibility of adding recovery to an existing implementation as well as evaluating the approach using the criteria outlined in section 3. Modification of an existing operational design represents a substantial initial test since, if it can be done successfully, it

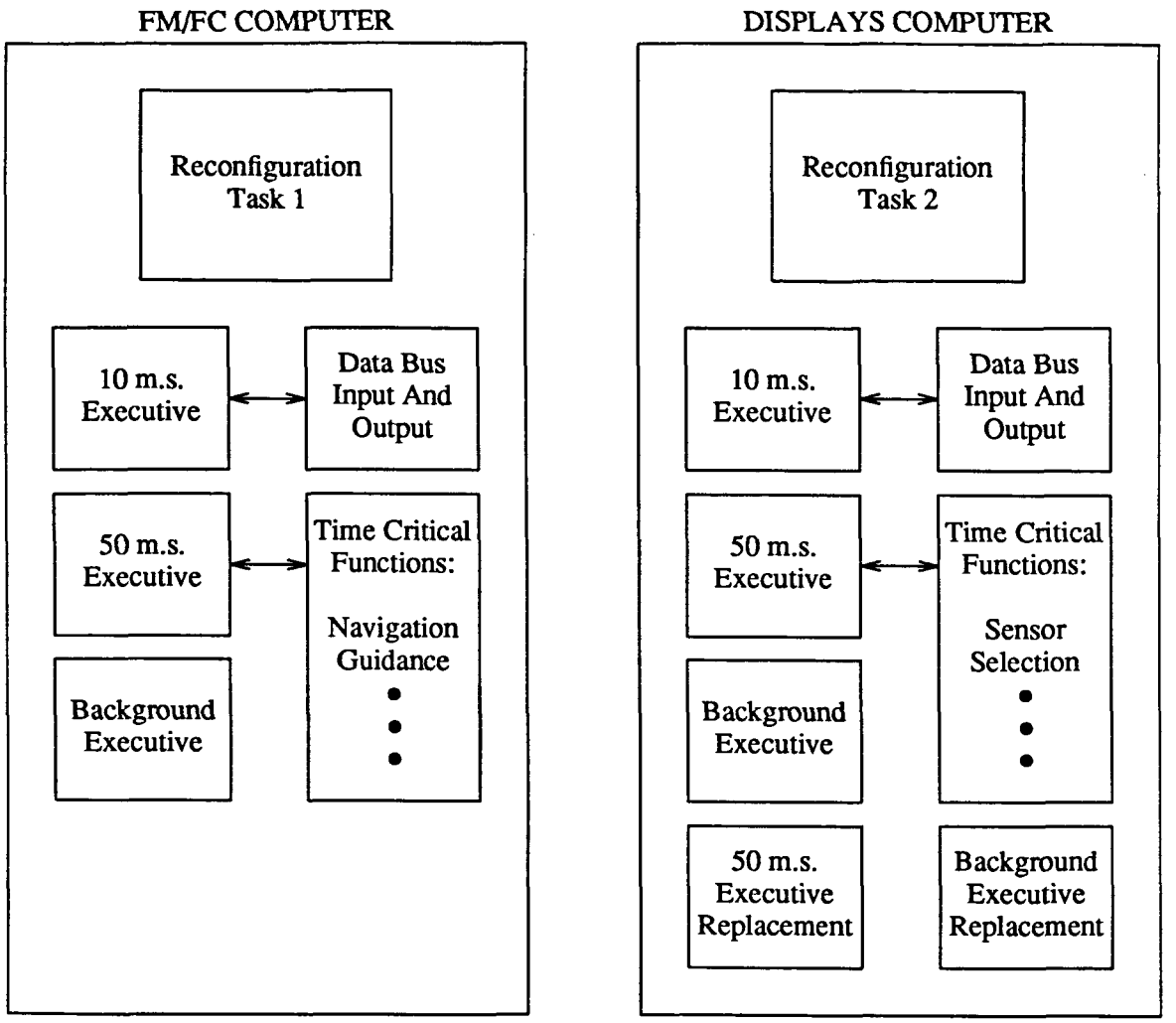


Fig. 1 - Software Structure

indicates that existing systems need not be discarded in order to take advantage of non-transparent continuation.

In a paper of this length it is not possible to document all the details of this analysis and so we present only a summary. The overall software structure is described followed by a more detailed examination of one component of the system. We then summarize the results of applying the various evaluation criteria.

Software Structure

The software architecture that we have designed for this target is shown in figure 1. During normal operation, each processor executes three tasks. One task on each processor executes every ten milliseconds and is synchronized by an external hardware clock. A second task on each processor executes every fifty milliseconds and is synchronized by entry calls from the associated ten-millisecond task. Finally, each processor has a background task that uses remaining processor time. Priorities are used to control the execution order of these tasks.

When both processors are operational, a block of data of between 200 and 3000 words, the length depending on whether certain new inputs are available, is transmitted from the FM/FC to the Displays computer every fifty milliseconds. The current sensor status is transmitted in the other direction at similar times.

The required recovery speed of the services being provided has determined the remainder of the software structure. The FM/FC functions must be resumed *very* quickly following failure. Despite the obvious inertia of a commercial air transport, certain control functions must be resumed in a few milliseconds. The functions provided by the Displays computer do not have to be recovered nearly so rapidly if it fails. These functions are interfacing with humans for the most part, either generating displays or accepting and processing inputs from keyboards and similar devices. Recovery within a few hundred milliseconds is adequate for these services.

The alternate software that takes over the services of the Displays computer when it fails is integrated into the FM/FC software. During each real-time frame, the FM/FC software checks the value of a flag that indicates status of the Displays computer. This flag is set by the FM/FC computer's reconfiguration task and is the only major action of that task. If the flag shows that the Displays computer is operational, the FM/FC software operates normally. If the computer has failed, different software is executed that provides reduced FM/FC service and skeleton display

and keyboard service. With this approach, it is possible that a considerable delay may occur between failure and resumption of display activity because the FM/FC software will only be aware of the failure after checking the flag. This may not occur until after extensive processing associated with FM/FC functions.

Since the FM/FC functions must be recovered rapidly, the approach of setting a flag following failure of the FM/FC computer and waiting for the Displays computer's software to check it is not sufficient. The software on the Displays computer cannot be designed like the FM/FC software just described. Our approach to ensuring very rapid recovery of the FM/FC functions is to locate on the Displays computer a skeleton version of parts of the FM/FC software. Also, complete replacements for the Display's computer's fifty millisecond and background executives reside on the Displays computer. There is no replacement for the Displays computer's ten-millisecond executive.

The reason for these replacement executives is to allow fifty-millisecond and background processing to be changed completely and very quickly on the Displays computer following failure of the FM/FC computer. These executives and all the alternate FM/FC software are normally idle. The executives contain entry definitions upon which they are normally suspended waiting for entry calls. Following failure, the replacement executives are started by entry calls from the reconfiguration task. The reconfiguration task also aborts the primary executives. The ten-millisecond executive is not replaced because its functions are, to a large extent, unchanged after failure. The reconfiguration task does set a flag so that this executive can make the necessary minor changes.

Since the reconfiguration task has the highest priority of all tasks on each machine, it is guaranteed to execute as and when required following failure. Thus the only delays between detection of the failure of the FM/FC computer and the resumption of FM/FC service on the Displays computer are (1) the time to start the reconfiguration task by generating an entry call, (2)

the time to start the replacement executives by entry calls from the reconfiguration task, and (3) the startup time of the replacement executives and associated computation functions. We expect these delays to be predicatable and small.

The skeleton FM/FC software is complete in that it can handle all the *critical* functions safely, although in some cases execution frequencies are reduced. Fortunately, it is not essential that all the control laws operate at the fastest real-time frame rate. Acceptable, although not perfect, performance is achieved at slower rates and so the replacement software operates at slower rates wherever possible.

Clearly, this is only one of many software structures that could be used. It is important to note, however, that the use of a skeletal replacement for the FM/FC software would not be feasible on any other target architecture. If the alternate software for the FM/FC functions had to be located on two or more processors, it would have to be partitioned in some way.

Microwave Landing System

As a detailed example of the way in which recovery might be provided for a critical function, we examine the Microwave Landing System software. The MLS system is an example of a function that we have viewed as critical although in a strict sense it is not. Failure of MLS service, even during landing, need not be disastrous provided that the pilot was informed and that he could fly the airplane manually. Taking over manually would be difficult if failure occurred close to the ground during an automatic landing but is possible. Recovery in this case could be limited to ensuring that sufficient functionality was available to allow manual control through the computer system.

Another view of MLS service is that it is sufficiently important that it should execute in a highly hardware-redundant computer such as a SIFT[12] or FTMP[13], in which the failure

probability is extremely low. However, although MLS processing could be performed on a highly reliable computer and is not essential following failure, for the purposes of this experiment, we have assumed that it is an example of a function that might be lost and has to be recovered. Were it operating in a weight or power restricted environment, such as a military aircraft or a spacecraft, it would not be possible to use extensive hardware replication to ensure safe operation of such functions. The MLS system represents a realistic example of the *type* of function that must be recovered in certain practical environments.

Although algorithmically the MLS system is complex, its overall structure is quite simple. Data is obtained from microwave receivers on the aircraft and used to compute position. Data acquisition requires confidence that a reliable source of MLS signals exists, as opposed to some form of radio noise. Thus the incoming data is checked over many real-time frames to ensure validity before any use is made of it. The data stream is filtered and various coordinate transformations are applied. The actual position, velocity and acceleration computations use a third-order complementary filter that depends on historic data for the various filter parameters. The MLS computations are performed every fifty milliseconds.

To recover MLS service if the FM/FC computer is lost, the alternate software must provide virtually identical computations and so the alternate software can be derived easily from the primary software. Fortunately, the MLS system can provide acceptable service if executed at half the normal rate and so the replacement software need be executed only every hundred milliseconds.

In order to start operating quickly, the replacement software must have available the status information about the incoming microwave signals and the historic information used as parameters in the filter. If this data is not consistent or not available, it can be recomputed in exactly the same way that the MLS software computes the information when it is initialized. The difficulty is that this computation takes several seconds since, for example, the determination that

the signals are reliable requires that they be monitored for many frames. To ensure that MLS service can be resumed immediately, the primary software must transmit its complement of data to the second processor on every real-time frame. For the MLS application, this amounts to a total of only approximately twenty real quantities.

Evaluation

Using the software structure just described for this application, most aspects of flexibility at development time are reduced very little by the application of non-transparent continuation. The reason is that the primary and alternate software components are mostly separate. Consider, for example, the Displays computer. It contains two systems that are almost completely isolated from each other, one for normal operation and one for operation after failure of the FM/FC computer. Since there are basically two systems, each can be dealt with separately and changes to either has little or no effect on the other. Of more importance is the impact that software changes on one processor have on the software of the other processor. However, again considering the Displays computer, changing the software used during normal operation has no impact on the software of the FM/FC computer. Clearly, changing the software that deals with failure of the FM/FC computer has no impact on the FM/FC computer software. This relatively slight impact on flexibility is certainly not typical and probably will not be found in the analysis of other architectures.

An area where flexibility may be affected considerably is the addition of a processor. The addition of a processor to a dual-processor system is a major change and will certainly require extensive software modifications. Our analysis of this element of flexibility is not yet complete.

Recall that the major issue at execution-time is overhead. In this implementation, the overhead is different on the two computers. Both have their memory requirements approximately doubled. The alternate software plus the space needed to store copies of important data is

roughly the same size as the original software.

Processing overhead on the Displays computer is minimal. The alternate software to deal with failure of the FM/FC computer adds nothing since that software is normally suspended. Similarly, the reconfiguration task adds nothing during normal operation. The overhead used in supporting the failure semantics is also slight since it is a function of the amount of inter-task communication between machines. In examining this communication for this application, we see that the rate is relatively low. Several rendezvous take place during each fifty millisecond frame but this frame rate is not great.

The processing overhead on the FM/FC is also minimal. The alternate software to deal with failure of the Displays computer adds very little since it is not executed during normal operation although the selection of which software is executed must be made on each cycle and so the overhead of a conditional operation per cycle is incurred. Similarly, the reconfiguration task adds nothing during normal operation. The overhead used in supporting the failure semantics is slight since it is basically the same as incurred by the Displays computer.

Providing consistent data is an area where overhead will vary widely with the application. It is quite possible that algorithms will require large amounts of data in order to operate correctly. In this application, however, this is not the case. Considering the MLS software as an example, the filter parameters and the MLS status information must be made consistent across machines on each frame. The total volume turns out to be less than one hundred bytes, however, and this imposes virtually no burden on processing time or the data communications bus. This order of data volume is typical of other functions in this application also.

SUMMARY AND CONCLUSIONS

In this paper, we have described an experiment that we are conducting in an attempt to evaluate the non-transparent approach to recovery in distributed systems programmed in Ada that must tolerate processor failure. Ada does not address the issues raised by processor failure in distributed systems and the analysis that we are performing is based on a version of Ada in which no syntactic changes have been made but necessary extensions to the semantics have been added.

The experiment involves analysis of a typical application consisting of the software for the flight control system in an experimental commercial air transport. Only the major control aspects of the software and the concurrent parts are being examined. The sequential computations are being ignored, except for their general characteristics such as execution time and code volume.

It is not possible to draw a general conclusion about the utility of non-transparent continuation from a single experiment such as this. Also, the experiment described here is ongoing. There are numerous criteria that have to be evaluated under a variety of conditions. We plan to examine only a subset of the conditions, and only part of that analysis is complete.

At this point in the experiment we are very encouraged by the analysis and feel that non-transparent continuation is useful for this application at least. Programs that incorporate non-transparent continuation are not as flexible during development as those that ignore continuation or rely on transparent continuation. However, we remain convinced that some form of continuation is essential in crucial distributed systems and that transparent continuation is impractical. Our preliminary analysis also indicates that the overhead experienced by this application in support of non-transparent continuation is not excessive.

ACKNOWLEDGEMENTS

It is a pleasure to acknowledge J.R. Williams, E.H. Senn and R.M. Hueschen of NASA Langley Research Center, and W.C. Clinedinst and D.A. Wolverton of Computer Sciences Corporation for many helpful technical discussions about the design and implementation of the ATOPS system. This work was supported in part by NASA grant NAG1-260.

REFERENCES

(1) Reference Manual For The Ada Programming Language, U.S. Department of Defense, 1983.

(2) Department Of Defense Requirements For High-Order Computer Programming Languages - STEELMAN, U.S. Department of Defense, 1978.

(3) J.C. Knight and J.I.A. Urquhart, "On The Implementation And Use Of Ada On Fault-Tolerant Distributed Systems", *IEEE Transactions On Software Engineering*, to appear (also available as University of Virginia Department of Computer Science Technical Report No. TR-86-19).

(4) D. Cornhill, "A Survivable Distributed Computing System For Embedded Applications Programs Written In Ada", *ACM Ada Letters*, Vol. 3, pp. 79-87, December 1983.

(5) D. Cornhill, "Four Approaches To Partitioning Ada Programs For Execution On Distributed Targets", *Proceedings of the 1984 IEEE Computer Society 1984 Conference on Ada Applications and Environments*, St. Paul, Minnesota, October 1984.

(6) N.G. Leveson and P.R. Harvey, "Analyzing Software Safety", *IEEE Transactions On Software Engineering*, Vol. SE-9, pp. 569-579, September 1983.

(7) J.C. Knight and S.T. Gregory, "A Testbed for Evaluating Fault-Tolerant Distributed Systems", Digest of Papers FTCS-14: *Fourteenth Annual Symposium on Fault-Tolerant Computing*, June 1984, Orlando, FL.

(8) P.A. Alsberg and J.D. Day, "A Principle For Resilient Sharing Of Distributed Resources", *Proceedings Of The International Conference On Software Engineering*, San Francisco, October 1976.

(9) J.N. Gray, "Notes On Database Operating Systems", in *Operating Systems: An Advanced Course*, Springer-Verlag, New York 1978.

(10) HAL/S Language Reference Manual, Intermetrics Corporation, Cambridge, MA.

(11) M.E. Rouleau, "Analysis Of Ada For A Crucial Distributed Application", M.S. Thesis, Department of Computer Science, University of Virginia, 1987.

(12) J.H. Wensley, et al, "SIFT, The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", *Proceedings of the IEEE*, Vol. 66, pp. 1240-1254, October 1978.

(13) A.L. Hopkins, et al, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor For Aircraft", *Proceedings of the IEEE*, Vol. 66, pp. 1221-1239, October 1978.

APPENDIX 4

CONCURRENT SYSTEM RECOVERY

CONCURRENT SYSTEM RECOVERY

Samuel T. Gregory

(Department of Computer Science, University of Virginia)

and

John C. Knight

(Department of Computer Science, University of Virginia)

9.1. Communicating Processes

Recovery blocks provide a mechanism for building backward error recovery into sequential programs. However, many programs such as operating systems and real-time control systems, are *concurrent.* A concurrent system consists of a set of communicating sequential processes. The processes execute in parallel and cooperate to achieve some goal. In so doing, they usually exchange data and synchronize their activities in time. Many concepts such as semaphores, monitors, ports, and rendezvous have been proposed to control the synchronization and communication of concurrent processes.

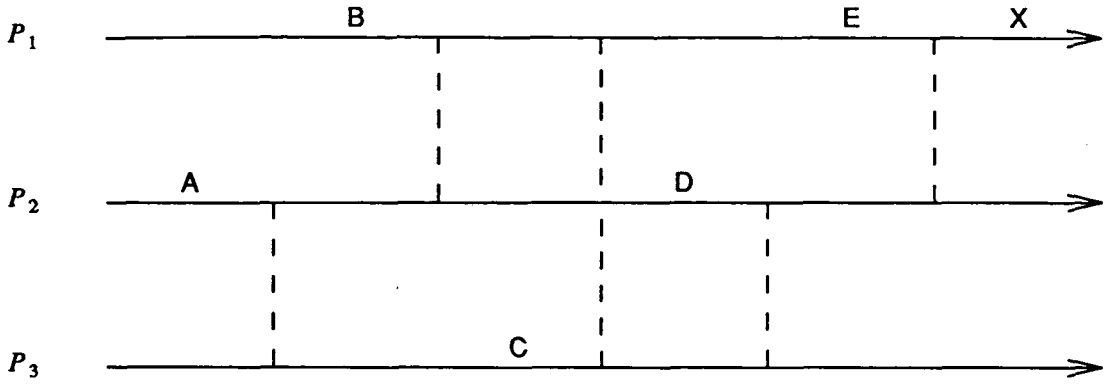
It is important that the technique of backward error recovery be extended to concurrent systems. These systems are very prone to error because they are usually extremely complex. The incorporation of fault tolerance may be a practical method of improving their reliability. Unfortunately, backward error recovery in concurrent systems cannot be provided merely by using recovery blocks in each separate process. Many new problems arise when concurrent

systems are considered and they are the subject of this chapter.

When two processes communicate, obviously information is passed between them. If two processes merely synchronize without explicitly passing data values, they still pass information. The information gained by each process in that case is (at least) that the other process has made a certain amount of progress. The utility of that kind of information depends upon the amount of knowledge about one process' design that was incorporated into the other process' design. Any form of synchronization, message passing, or shared variable update allows information to pass from one process to another.

Suppose two processes communicate between the time that a fault in one of them produces the first error and the time that an error is detected. Since the information transfer is two-way, whichever process has developed the error may have spread that error to the other. Further, the error might not be detected by the process containing the fault. A solution to these problems is to roll back, i.e. perform backward error recovery, on both processes. If the recovery points for all of the processes involved are not carefully coordinated, a problem called the *domino effect* [37] could result.

Figure 9.1 illustrates the domino effect with three processes P_1, P_2 and P_3 progressing with time to the right. Suppose each process has established recovery points at arbitrary times shown as the letters A, B, C, D, and E in the figure. The vertical dashed lines represent communications between the processes. An error detected in P_1 at X causes process P_1 to be rolled back to E and process P_2 to be rolled back to D. But this rollback invalidates information exchanged between P_2 and P_3, so P_3 must also be rolled back. The closest recovery point for process P_3 is C. Since P_1 and P_3 communicated between points C and E, P_1 must again be rolled back, this time to B. The effect could conceivably spread to other processes and continue from recovery point to



The Domino Effect

Figure 9.1

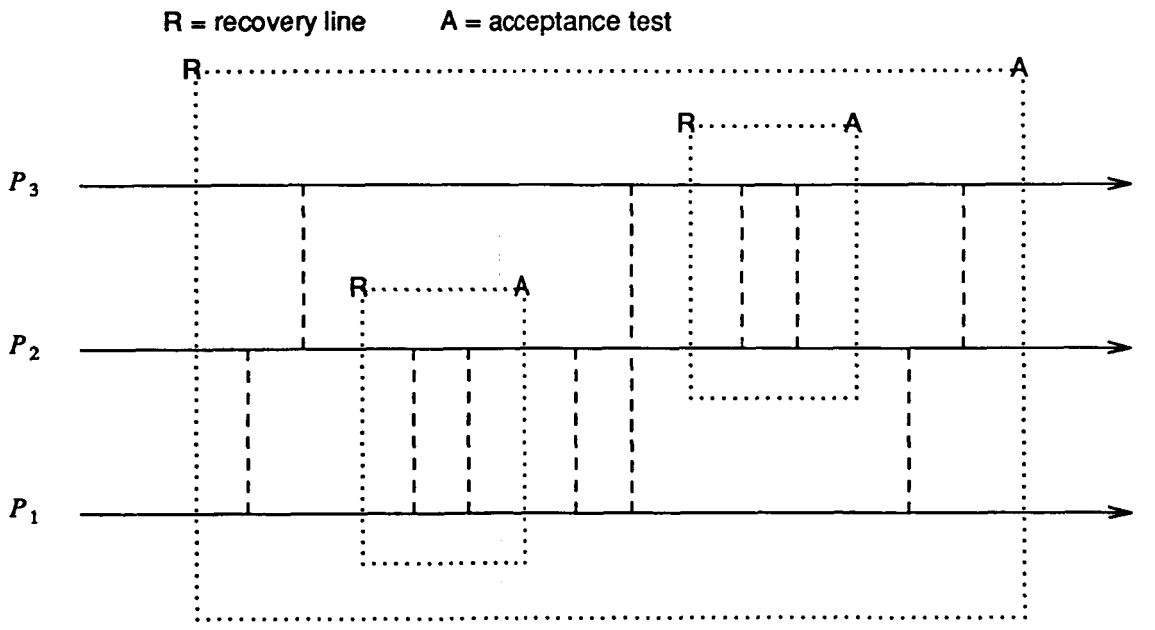
recovery point (like falling dominos) until the entire software system was rolled back to its initial state, thus discarding all information gathered during its operative life.

Clearly, the domino effect has to be avoided. To do this, the backward error recovery method employed must coordinate the establishment of recovery points for communicating processes and limit the "distance" they can be rolled back.

The domino effect is the major problem in the provision of backward error recovery in concurrent systems. But unrecoverable objects have to be dealt with just as they do in sequential systems, and other subtle problems of program structure arise. In this chapter we describe various methods of dealing with the domino effect including *conversations, exchanges, FT-Actions, dialogs,* and *colloquys.* We also describe two other approaches to dealing with recovery of concurrent systems, *chase protocols,* and *spheres of control.* Finally, we summarize the problems of program structure.

9.2. Conversations

The *conversation* [37] is the canonical proposal for dealing with communicating processes via backward error recovery. In a conversation, a *group* of processes agree about when recovery points will be created and discarded. Each may create a recovery point separately, but they must synchronize the time at which the recovery points are discarded. The set of recovery points is referred to as a recovery line. Only processes within the group may communicate. At the end of their communication, which may include the passage of multiple distinct sets of information, they each wait for the others to arrive at an acceptance test for the group. If they pass the acceptance test, they *commit* to the information exchange that has transpired by discarding their recovery line and proceeding. Should they fail the acceptance test, they all restore their states from the



Three Nested Conversations

Figure 9.2

recovery line. No process is allowed to *smuggle* information out or in by communicating with a process that is not participating in the conversation's organization.

Conversations may be nested. From the point of view of a surrounding conversation, a nested conversation is an atomic action. The encased activity seems either not to have begun or to have completed, and no information that would be evidence to the contrary escapes.

Figure 9.2 shows an example in which three processes communicate within one conversation and two subsets of two each communicate separately within two nested conversations. The dotted rectangles represent the recovery lines on the left verticals, the acceptance tests on the right verticals, and the prohibition of smuggling on the horizontal portions.

The recovery lines are shown as simultaneously established, but that is not required. Note that, if an error were detected in process P_1 while processes P_2 and P_3 were conversing, all effects since the larger recovery line (including the already-completed conversation between P_1 and P_2) would be undone. Once individually rolled back and reconfigured, the same set of conversant processes attempt to communicate again, and eventually reaches the same acceptance test again. Also any other failure of one of the processes is equivalent to a failure of the acceptance test by all of them. Thus, a conversation is a kind of parallel recovery block where each of the primary and the alternates are execution segments of a set of processes.

Conversations were originally proposed as a structuring or design concept without any syntax that might be used in a practical programming language. The *Name-Linked Recovery Block* was proposed by Russell as a syntax for conversations [38]. The syntax appropriates that of the recovery block:

CONV <conversation identifier> : <recovery block>

What would otherwise be a recovery block within a process, becomes part of a conversation by associating a name with the recovery block. The name is called the conversation identifier, and all processes executing recovery blocks with the same conversation identifier become members of the conversation. The primary and alternate activities of the recovery block become that process' primary and alternate activities during the conversation, and the recovery block's acceptance test becomes that portion of the conversation's acceptance test appropriate to this process. The conversation's acceptance test is evaluated after the last member of the conversation reaches the end of its primary or alternate. If any of the processes fail their acceptance tests, all conversants are rolled back.

In other work [39], Russell proposed loosening the structure of conversations. He proposed that the establishment, restoration, and discard of recovery points for processes be under the dynamic control of the applications' programmer rather than encased in a rigid syntax. He gave three primitives for these operations: MARK, RESTORE, and PURGE respectively. They are all parameterized to designate the subject recovery point and apply to an individual process. This allows the programmer to save many states and restore the one he chooses, rather than the most recent. Recovery points are not constrained to be RESTOREd in the reverse of the order MARKed. The proposal assumes message buffers for inter-process communication. As part of backing a process up to a recovery point, previously received messages are placed back into the message buffers.

This mechanism ignores the possibility that the information within a message can contaminate a process' state. Such an approach only applies to producer-consumer systems. Many concurrent systems are feedback systems. A producer almost always wants to be informed about the effects of the product, and a consumer almost always wants to have some influence over

what it will be consuming in the future. The relationships between sensors and a control system and between a control system and actuators can be viewed as pure producer-consumer relationships, but sensors and actuators are more accurately modeled as unrecoverable objects. The proposal allows completely unstructured application of the MARK, RESTORE, and PURGE primitives. This fact, along with the complicated semantics of conversations, which the primitives are provided to implement, affords the designer much more opportunity to introduce faults into the software system. For example, the use of the PURGE primitive on a recovery point represents a "promise" never to use a RESTORE primitive on that recovery point. There is no enforcement of this "promise". Also, the utility of the ability to save two recovery points A and B and later restore A before restoring B is unclear.

Kim has proposed several syntaxes for conversations [24]. His approaches assume the use of monitors [14] as the method of communication among processes. In the *Conversation Monitor*, shown in Figure 9.3, the conversing activities are grouped with their respective processes' source code, but are well marked at those locations. In the *Conversation Data Type*,

---

```
ENSURE <boolean expression>
USING-CM <conversation monitor identifier>
    { <conversation monitor identifier> }
BY
  <primary>
ELSE BY
  <alternate 1>
...
ELSE BY
  <alternate n>
ELSE ERROR
```

Kim's Conversation Monitor Syntax

Figure 9.3

---

shown in Figure 9.4, the conversing actions of the several processes are grouped into one place so that the conversation has a single location in the source code. The issue these variations address is whether it is better to group the text of a conversation and scatter the text of a process or to group the text of a process and scatter the text of a conversation.

```
TYPE c = CONVERSATION( <conversation names > )
      PARTICIPANTS proca( <formal parameters> );
                   procb( <formal parameters> );
                   ...
      VAR cm1 : <conversation monitor type> ;
        cm2 : <conversation monitor type>;
        ...
      ENSURE <acceptance test> BY
        BEGIN proca : <statements>
              procb : <statements>
              ...
        END
      ELSE BY BEGIN
          proca : <statements>
          procb : <statements>
          ...
      ...
      ELSE ERROR
      BEGIN
        INIT cm1,cm2...
      END

VAR conv1 : C;

          (a)

conv1.proca( <actual parameters> );

          (b)
```

Kim's Conversation Data Type Syntax

Figure 9.4

Kim's third scheme, the *Concurrent Recovery Block* shown in Figure 9.5, attempted to resolve the differences between the first two by enclosing the entirety of the processes within the conversation. Here, a conversation is a special case of a recovery block, within a single parent process, in which the primary and the alternates consist solely of initializations of monitors and activations of processes.

The concurrent recovery block is not really a construct for programming concurrent systems. Rather, it is a construct for programming sequential systems in which a particular execution order for occasional statement sequences is not required.

```
ENSURE <boolean expression> BY BEGIN
  INIT monitor.1;
  ...
  INIT process1.1( <actual parameters> );
  INIT process2.1( <actual parameters> );
  ...
  END
ELSE BY BEGIN
  INIT monitor.2;
  ...
  INIT process1.2( <actual parameters> );
  INIT process2.2( <actual parameters> );
  ...
  END
 ...
ELSE BY BEGIN
  INIT monitor.n;
  ...
  INIT process1.n( <actual parameters> );
  INIT process2.n( <actual parameters> );
  ...
  END
ELSE ERROR
```

Kim's Concurrent Recovery Block Syntax

Figure 9.5

None of Russell's or Kim's conversation schemes enforce the prohibition against smuggling. If processes use monitors, message buffers, or ordinary shared variables, other processes can easily "reach in" to examine or change values while a conversation is in progress. The conversation monitor is designed to prevent smuggling but, as Kim's description stands, it allows a problem that is even more insidious than smuggling. A monitor used within a conversation is initialized for each use of the conversation, but not for each attempt within a conversation. This allows partial results from the primary or a previous alternate to survive state restoration within the individual processes. Since such information is in all probability erroneous, it is likely to contaminate the states within all subsequent alternates.

A major difficulty of the conversation scheme and of all its follow-up syntactic proposals lies in the acceptance test(s). The strategies involved in the primary and in the many alternates may be so divergent as to require separate checks on the operation of each "try" as well as an overall check for acceptability as regards the goal of the statement.

Another difficulty involving acceptance tests appears when we consider that each process in a conversation has its own individual reasons for communicating, while the system of which these processes are a part has more global concerns for bringing them together. A single, monolithic acceptance test would be too concerned with acceptability in terms of the surrounding system to detect errors local to the component processes. Similarly, the combination of local acceptance tests of the individual processes is insufficient since it does not incorporate the design of the surrounding system. A conversation needs a check on satisfaction of the surrounding system's goal in the communication as well as checks on satisfaction of the component processes' goals.

Desertion is the failure of a process to enter a conversation when other processes expect its presence. Whether the process will never enter the conversation, is simply late, or enters the conversation only to take too long or never arrive at the acceptance test(s), does not matter to the others. The processes in a conversation need a means of extricating themselves if the conversation begins to take too long. Each process may have its own view of how long it is willing to wait, especially since processes may enter a conversation asynchronously. Only the concurrent recovery block scheme addresses the desertion problem. The solution there is to enclose the entirety of each participating process within the conversation. This is too restrictive in that not only cannot a process fail to arrive at a conversation, it cannot exist outside of the conversation.

The original proposal of conversations made no mention of what was to be done if the processes ran out of alternates. Two presumptions may be made (1) that the number of alternates is unbounded, or (2) that an error is to be detected automatically in each of the processes, as is assumed in all of the proposed syntaxes. What the syntactic proposals do not address is that, when a process fails in a primary attempt at communication with one group of processes to achieve its goal, it may want to attempt to communicate with an entirely different group as an alternate strategy for achieving that goal. This is the kind of divergent strategy alluded to above. The name-linked recovery block and the conversation monitor schemes do not mention whether it is an error for different processes to make different numbers of attempts at communicating. Although those schemes may assume that is covered under the desertion issue, it may not be if processes are deliberately allowed to converse with alternate groups.

It can occur that a nested conversation commits to a change in an unrecoverable object only to have the surrounding conversation fail. This presents a problem. One suggestion was that the object be marked for alteration but that the change not actually occur until the outermost

conversation commits [27].

How to construct meaningful acceptance tests was an open problem for recovery blocks. It remains so for conversations. An acceptance test must be able to detect errors in results of any alternate in the context of independently constructed algorithms. Yet the same acceptance test must be able to pass results of any alternate, no matter how degraded the service it provides. The test must not be so complex or slow as to duplicate the algorithms of the primary or alternates. Although some thought has been given to this problem [27], it too remains open.

9.3. Exchanges and Simple Recovery

Many real-time systems are concurrent and are used frequently in applications requiring very high reliability. Real-time systems using a cyclic executive have a relatively simple structure which can be used to advantage in implementing backward error recovery.

Under a cyclic executive, time is divided into "frames". Inputs are accepted at the beginning of each frame, and outputs are produced at the end of each frame. Anderson and Knight proposed *exchanges* [2] in an attempt to adapt conversations to this real-time program structure.

An exchange is a conversation in which all of the communicants are created at the recovery line and destroyed at the acceptance test. The beginning of a frame represents the "recovery line", and the acceptance test is at the end of the frame. Failure of the acceptance test causes alternate outputs to be generated for the current frame using some simple alternate computation, e.g. repeating those of the previous frame. The only information saved at the "recovery line" is that needed to provide the alternate outputs since the communicating processes will be started

anew rather than backed up. The execution-time support keeps track of which processes fail individually and how often the group fails. This information transcends frame boundaries and is used to determine when a process is to be replaced for the next or subsequent frames.

The idea of exchanges has direct utility only in systems employing the cyclic executive scheduling regime. The proposal does not address systems of fully asynchronous processes or systems employing mixed disciplines. The exchange concept thus imposes a cobegin ... coend programming structure, which may not always be suitable. For example, it becomes difficult to program multiple frame rate systems, the first variation that is often imposed on the cyclic executive theme [31].

9.4. Deadlines

The *Deadline Mechanism* was proposed by Campbell, et al to deal with timing faults in real-time systems [8]. When a goal must be achieved before a certain amount of time passes, a preferred algorithm is supplied along with an alternate algorithm and a duration. The alternate algorithm is assumed to be correct and deterministic so the amount of time it requires is known *a priori*. The underlying scheduler is responsible for ensuring that, if the preferred algorithm cannot be completed before the deadline (duration plus time the preferred algorithm started), then the alternate algorithm can be. Several simulation studies have been performed showing a reduction in timing failures when such a mechanism is employed [8,49,29].

The deadline mechanism *assumes* that the alternate algorithm is correct. Nothing is said about checking the acceptability of the preferred algorithm's results if it does complete on time. The proposal assumes that the amount of time required by the alternate algorithm is known

a priori, yet provides no method of communicating this information to the underlying scheduler. The additional (alternate) processes in the scheduling mix could even be the cause of a failure of a preferred algorithm to complete on time. No mention is made of how the data states of the preferred and alternate algorithms are to be kept separate. This proposal focuses too narrowly upon only one issue, that of timing, and provides incomplete coverage of that.

9.5. Chase Protocols

Some concurrent systems do not require the sender of a message to wait for message receipt. In some of these systems, a message can be "in transit" for long periods of time. In such systems, backward recovery in the sender may require that the message be "chased down" and removed or, if already received, that the message's effects be undone. For systems such as these, the idea of *chase protocols* was invented [33].

As a process backs up to a recovery point, all messages which it has sent since establishing that recovery point are chased down. Messages caught in transit are simply deleted. If a message has already been received, the receiving process is backed up to the most recent recovery point it established before it received the message. The receiving process then enters a chase protocol to deal with messages it had sent since establishment of the recovery point.

Also, as a process backs up to a recovery point, all messages which it has received since establishing that recovery point are gathered for replaying. For those messages which are unrecoverable, e.g. the message was issued in response to a message that has been retracted, the senders are made to back up and enter the chase protocol. A chase protocol terminates when a recovery line is found dynamically.

For cases in which data exists independently of any process, the data items themselves "send" and "receive" the special fail messages required to chase down other information. Copies of the data are considered to have been sent to processes as messages, and for updating purposes, back from processes to the data items themselves. It is under these circumstances that the recoverability of messages that might otherwise be replayed at a process becomes important. If a copy-of-data message is not recoverable, the data item must be backed up by backing up the processes responsible for its current value to recovery points beyond their setting of that value.

Chase protocols work on the assumption that the consequences of the domino effect will usually be limited, and that very extensive rollback is pathological. Rather than attempting to prevent the domino effect explicitly, they attempt to find a recovery line by systematic search. Thus, the most obvious and damning drawback of chase protocols is that they leave a system open to the possibility (perhaps remote) of the domino effect. This may be unacceptable in critical applications.

9.6. Spheres of Control

Davies catalogued many of the concepts of concurrent systems, recovery, and integrity in a taxonomy he called *data processing spheres of control* [11]. Spheres of control are intended to address many problems such as keeping processes from interfering with each other, backing processes to a previous state, and preventing other processes' use of uncommitted data. The concepts allow for multiple processes to cooperate within recovery regions while describing the restrictions on their activities necessary for maintaining integrity within such an environment. These multiple processes may be (largely) independent, but may be using partial (uncommitted) data from each other. Spheres of control can cross machine boundaries; one of the examples

given is of remote procedure call, but predates the term.

Spheres of control make use of the concepts of process atomicity, commitment, recovery before a process has committed, recovery after a process has committed, and maintaining consistency by controlling dependence of processes' activities on those of others.

The concepts were described without implementation advice for generality of application. Indeed, the description can be considered a taxonomy or catalogue of techniques already used in some systems. The emphasis is on placement, or what needs to be done in a system to ensure integrity and recoverability, without prescribing how.

The descriptions are in terms that might be used by accounting auditors of business-oriented applications.

As a catalogue of ideas, without an enforceable basis for their application, spheres of control are rather disorganized. However, Davies' concluding remark was that many of those ideas need to be included in a programming language to permit their use and enforcement in applications.

9.7. FT-Actions

All of the other approaches described in this chapter attempt to provide backward recovery. The Fault-Tolerant Atomic Action (FT-Action) introduced by Jalote and Campbell [20] (also known as the S-Conversation [19]) is an attempt to unify the concepts of backward and forward recovery for concurrent systems. Backward recovery is provided using conversations and forward recovery by a systematic approach to exception handling combined with atomic actions.

All of the concepts in the FT-Action are introduced as extensions to the language CSP [17].

Central to the theme of FT-Actions is a revised form of atomic actions. Jalote and Campbell distinguish between the original definition of atomic action in which atomicity is combined with state restoration and a reduced concept in which no state restoration takes place. The former they refer to as *recoverable atomic actions* and the latter as *basic atomic actions*. Both concepts are required since the former implies backward error recovery. To allow for forward recovery, the more fundamental notion is used.

The FT-Action is defined in terms of the language CSP because CSP provides a particularly simple framework in which to study concurrent systems. The language has no shared memory between processes, and all inter-process communication must be programmed explicitly. These simple semantics eliminate most potential forms of smuggling and constrain communication.

As with all the other proposals discussed in this chapter, when used for backward recovery an FT-Action is basically a construct for forcing processes to communicate in an orderly fashion to prevent the domino effect. In general, processes may only communicate within an FT-Action and then only with other processes in the same FT-Action. FT-Actions may be nested to provide multiple recovery regions.

If backward error recovery is required, the syntax of the FT-Action provides a notation for describing conversations. The processes participating in the FT-Action are listed in a declaration and each process describes its primary and alternate modules in a recovery-block-like syntax. For any given FT-Action each participant is required to have the same number of alternates.

The processes execute their primaries, communicating with each other as necessary, and then evaluate their acceptance tests. If any test fails an exception is raised, but an exception may also be raised at any point by any process to signal failure during execution of its primary. The FT-Action completes if all acceptance tests are successful. If they are not, all processes back up and try the next alternate. If the alternates are exhausted without success, an exception is raised in the surrounding block (if there is one) to signal that the entire FT-Action has failed.

If forward error recovery is to be used, the FT-Action for each process describes the code sequence that the process will attempt together with an exception handler. Failure of the attempt is signaled by the process raising an exception and, in that case, the exception is raised in *all* the processes which then all execute exception handlers. Forward and backward recovery are combined by allowing any alternate in a backward-error-recovery structure to contain an exception handler. If an exception is raised and a handler is present, the handler deals with the situation if it can. If no handler exists, or a further exception is raised in a handler, then backward error recovery is invoked.

The mappings of the various forms of the FT-Action into CSP primitives are given by Jalote and Campbell. They point out that these mappings could be implemented easily in a preprocessor thereby allowing programs written in CSP enhanced with FT-Actions to be translated into CSP and thereby executed.

In practice, there are several issues that FT-Actions do not address. For example, there is no explicit provision for dealing with deserter processes. The designers of the concept acknowledge the problem, and point out that some form of time-out needs to be included. In addition, as will be shown later, the use of the original conversation mechanism limits the diversity that the can be achieved in the alternates and the coverage of the acceptance tests.

9.8. Dialogs and the Colloquy

In an effort to solve the general problems associated with conversations as discussed in section 9.2, Gregory and Knight developed the *dialog* and *colloquy* [12,13]. These concepts permit true independence of algorithms between alternates, allow time constraints to be specified, and are accompanied by syntactic proposals that are extensions to the language Ada.

A dialog is a way of enclosing a set of processes in an atomic action. A colloquy is a construct in which a set of atomic actions (specified by dialogs) can be described. From the perspective of each process, the set of atomic actions in which it participates constitutes the primary and the series of alternates of a fault-tolerant structure.

Further flexibility is introduced in these concepts by providing both a local acceptance test for each process and a global acceptance test for the group.

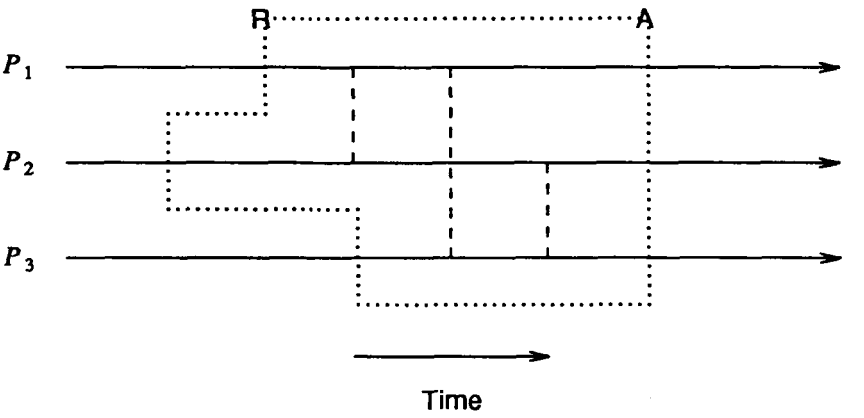
9.8.1. Dialogs

In a *dialog*, a set of processes establish individual recovery points, and communicate among themselves and with no others. They then all either discard their recovery points or restore their states from their recovery points, and then proceed.

Success of a dialog is the determination that all participating processes should discard their recovery points and proceed. *Failure* of a dialog is the determination that they should restore their states from their recovery points and proceed. Nothing is said about what should happen *after* success or failure; in either case the dialog is complete.

Dialogs may be properly nested, in which case the set of processes participating in an inner dialog is a subset of those participating in the outer dialog. Success or failure of an inner dialog does not necessarily imply success or failure of the outer dialog. Figure 9.7 shows a set of three processes communicating within a dialog.

The *discuss* statement is the syntactic form that denotes a dialog. Figure 9.8 shows the general form of a discuss statement. The *dialog_name* associates a particular discuss statement



Three Processes Communicating in a Dialog

Figure 9.7

```
DISCUSS dialog_name BY
    sequence_of_statements
TO ARRANGE Boolean_expression;
```

A DISCUSS Statement

Figure 9.8

with the discuss statements of the other processes participating in this dialog, thereby determining the constituents of the dialog *dynamically*. At execution time, when control enters a process' discuss statement with a given dialog name, that process becomes a participant in a dialog. Other participants are any other processes which have already likewise entered discuss statements with the same dialog name and have not yet left, and any other processes which enter discuss statements with the same dialog name before this process leaves the dialog. Either all participants in a dialog leave it with their respective discuss statements successful, or all leave with them failed, i.e. the dialog succeeds or fails.

The Boolean expression in the discuss statement is the local acceptance test. It represents the process' *local* goal for the interactions in the dialog. If this Boolean expression or that in the corresponding discuss statement of any other process participating in this dialog is evaluated **false**, the discuss statement of each participant in the dialog *fails*. If all of the local acceptance tests succeed, the common goal of the group, i.e. the *global* acceptance test is evaluated. If this common goal is **true**, the corresponding discuss statements of all participants in the dialog succeed; otherwise they fail. Syntactically, the common goal is specified by a parameterless Boolean function with the same name as the dialog name in the discuss statement.

For the actions of the dialog's participants to appear atomic to other processes, all forms of communication must be controlled. The set of variables shared by processes participating in a dialog are locked by the compiler and execution-time support system to prevent smuggling. While locked, the shared variables may only be used by processes in that dialog. Which variables are to be shared, and therefore locked, is specified in *dialog declarations*. The dialog names used in discuss statements are also declared in dialog declarations. The general form of a dialog declaration is:

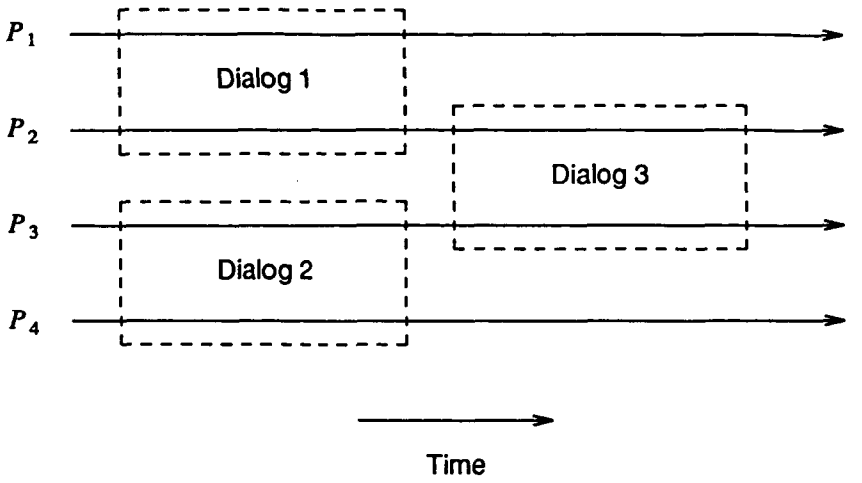
```
DIALOG function_name SHARES ( name_list );
```

The *function_name* is the identifier being declared as a dialog name and is the name of the function defining the global acceptance test. The names in the *name_list* are the *shared* variables which will be used within dialogs that use this dialog name.

9.8.2. The Colloquy

A colloquy is a collection of dialogs. At execution time, a dialog is an interaction among processes. Each individual process has its own *local* goal for participating in a dialog, but the group has a larger *global* goal; usually providing some part of the service required of the entire system. If, for whatever reason, any of the local goals or the global goal is not achieved, a backward error recovery strategy calls for the actions of the particular dialog to be undone. In attempting to ensure continued service from the system, each process may make another attempt at achieving its original local goal, or some *modified* local goal through entry into a *different* dialog. Each of the former participants of the now defunct dialog may choose to interact with an *entirely separate* group of processes for its alternate algorithm. The altered constituency of the new dialog(s) almost certainly requires new statement(s) of the original global goal. The set of dialogs which take place during these efforts on the processes' part is a *colloquy*. A set of four processes engaged in a colloquy that involves three dialogs is shown in Figure 9.9.

A colloquy, like a dialog or a rendezvous in Ada, does not exist syntactically but is entirely an execution-time concept. However, the places where the text of a process statically indicates entry into colloquys are marked by a variant of the Ada select statement called a *dialog_sequence*.



Four Processes in a Colloquy of Three Dialogs

Figure 9.9

The general form of a dialog_sequence is shown in Figure 9.10. At execution time, when

```
SELECT
    attempt_1
  OR
    attempt_2
  OR
    attempt_3

  TIMEOUT simple_expression
    sequence_of_statements

  ELSE
    sequence_of_statements
END SELECT;
```

Dialog_Sequence

Figure 9.10

control reaches the select keyword, a recovery point is established for that process. The process then *attempts* to perform the activities represented in Figure 9.10 by attempt_1. The attempt is actually a discuss statement followed by a sequence of statements. If the performance of these activities is *successful*, control continues with the statements following the dialog_sequence. If the attempt was not successful, the process' state is restored from the recovery point and the other attempts will be tried in order. Thus, the dialog_sequence enables the programmer to provide a primary and a list of alternate algorithms by which the process may achieve its goals at that point in its text. Note however that the process may communicate with entirely different sets of processes in each attempt, thereby allowing greater diversity in the alternates than is possible in the conversation or similar. Also, each process may specify a different number of alternates from the other processes to accommodate its own goal.

Exhaustion of all attempts for a given process with no success brings control to the **else** part after restoration of the process' state from the recovery point. The **else** part contains a sequence of statements which allows the programming of a "last ditch" algorithm for the process to achieve its goal. If this sequence of statements is successful, control continues after the dialog_sequence. If not, or if there was no statement sequence, the surrounding attempt fails.

Timing constraints can be imposed on colloquys (and hence on dialogs). Any participant in a colloquy can specify a timing constraint which consists of a simple expression on the **timeout** part of the dialog_sequence. A timing constraint specifies an interval during which the process may execute as many of the attempts as necessary to achieve success in one of them. If the interval expires, the current attempt fails, the process' state is restored from the recovery point, and execution continues at the sequence of statements in the **timeout** part. The attempts of the other processes in the same dialog also fail but their subsequent actions are determined by their own dialog_sequences. If several participants in a particular colloquy have timing constraints,

expiration of one has no effect on the other timing constraints. The various intervals expire in chronological order. As with the else part, the timeout part allows the programming of a "last ditch" algorithm for the process to achieve its goal, and is really a form of forward recovery since its effects will not be undone, at least at this level.

The dialog and colloquy concepts provide implementable answers to the difficulties of other backward error recovery proposals. These ideas afford the error detection flexibility of multiple acceptance tests. They also invert the relationship between operation of the recovery point and inter-process communication. This permits truly independent alternate algorithms to the extent that a process can communicate with different groups of processes to achieve its goals.

Colloquys avail the programmer of many powerful facilities for management of backward error recovery. It is tempting to think that this solves all the problems that might arise, and that the syntax for the colloquy can be integrated into a language for programming concurrent systems with no further concern.

9.9. New Difficulties

Problems beyond the domino effect arise when including recovery in realistic concurrent systems [13]. They have to do with enforcement of the prohibition on smuggling and organization of programs.

The merging of recovery facilities into a real language can reveal semantic difficulties not readily apparent in the general discussion of the ideas. Certain aspects of actual programming languages seem to conflict with the goals and design of backward error recovery facilities. In this section, we introduce some of the problems which arise in attempting to merge backward error

recovery into a modern programming language. This examination discloses several new problems with backward error recovery in real languages. These problems arise because of the fundamental requirements of backward error recovery in concurrent systems. We use the dialog and colloquy merely as examples.

In their most general form, the problems are:

(1) the many means of *smuggling* of information that are afforded by many programming language constructs, and

(2) the incompatibilities between the planned establishment of recovery lines for backward error recovery and the existing explicit communication philosophies of modern programming languages.

9.9.1. Smuggling

Smuggling is a transfer of information, or communication, between a process engaged in a particular dialog and a process not so engaged. From the point of view of a surrounding dialog, a nested dialog is supposed to be an atomic action. The encased activity seems either not to have begun or to have completed, and no information that would be evidence to the contrary escapes. Were smuggling allowed, backward recovery of the participants in a dialog could produce an inconsistent state. Thus smuggling must be prevented.

We have so far ignored the many means of smuggling. Smuggling is usually *assumed* to be controllable. All of the approaches mentioned in this chapter depend for their avoidance of the domino effect upon the prohibition of smuggling. The very term "sphere of control" evokes an

image of a barrier surrounding the communicating processes and their uncommitted results. The FT-Action was defined in an language without means of smuggling, so its presentation ignored the issue.

Many means of smuggling exist in modern programming languages. They break down into explicit and implicit information flows. Explicit information flows derive from deliberate communications attempts on the part of the programmer using the explicit communications mechanisms in the language such as messages or rendezvous. Implicit information flows occur through shared variables, attributes and process manipulation.

A major potential form of smuggling lies in message traffic. In Ada, smuggling through explicit information flows, is not problematic. The Ada rendezvous is a specialized form of message communication through a restricted set of protocols. When a process attempts to communicate with another, it is suspended until the communication is complete. The sender does not proceed immediately after sending a message. This is the only form of explicit communication in Ada. The dialog prevents smuggling via messages for an Ada-like language. A more general message-based language would present more problems for backward error recovery.

The second form of smuggling, that through implicit information flows, is much more involved. *implicit* information flows are methods by which one process gains information about another process' activities or status without using the explicit communications statements provided in the language. Implicit flows come in two categories. The first category is provided by the facilities in a language which one would normally expect to allow implicit information flows. The other category is provided by language facilities or features which one would not normally think of as involving communication.

The first category, expected implicit information flows, is represented by shared, variable objects. One normally expects implicit information flows through these objects. They come in two sub-categories, based upon their modes of access. *Shared variables* are objects with one access path. *Aliasing* and *pointers* provide objects with multiple access paths.

The category of unexpected implicit information flows is represented by process manipulations. Ada allows processes to be manipulated in several ways. These are task creation, task destruction, and examination of other processes' execution states. This last one is represented by Ada's task attributes. The dynamic creation and destruction of processes are facilities which one would not expect to afford implicit information flows. That smuggling may occur through them is a very unusual concept.

9.9.2. Communication Philosophies

The second of the most general problems is the existence of incompatibilities between the planned establishment of recovery lines for backward error recovery and the explicit communication philosophies of modern programming languages. These stem from conflicts between the planned establishment of recovery lines and modern programming precepts. These incompatibilities are typified by detailed problems with service tasks in Ada. Some of them are recapitulated here.

First, Ada allows a task to make nondeterministic choices among entries when accepting calls. There is no corresponding nondeterminism when choosing to enter a dialog. Second, Ada enforces mutual exclusion among entry calls being serviced. The dialog allows any process to enter the communication at will. Third, a server task may be requested to perform its service at

any time in Ada. Under the dialog regimen, it seems a server must actively seek out its clients to achieve the same dialog nesting. Finally, the server cannot leave a dialog after dealing with one client and before seeking the next client until the first client is ready to leave (i.e. the server can become trapped).

Ada has nondeterminism and exclusivity in its communication mechanism. The dialog, which forms an envelope around communication, is not nondeterministic. The envelope restricts severely one's use of nondeterminism. The envelope is also intentionally non-exclusive to participants. These program structuring problems are not specific to the dialog and colloquy concepts. Rather, they represent a general conflict of planned establishment of recovery lines and languages designed to facilitate use of modern programming precepts.

9.9.3. Summary

The language facilities shown in this chapter seem on the surface to be adequate for recovery in concurrent systems, however they turn out to be incomplete solutions to these problems. A formal approach to recovery in concurrent systems should have syntactic expression so its semantic rules may be enforced automatically. The approach and its syntax cannot be designed separately from other facilities of the programming language into which they are to be included. To avert interaction of facilities that might allow subversion of the recovery approach's rules, the language must be designed with recovery in mind from the outset.

References

(1) *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A (22 January 1983).

(2) T. Anderson and J. C. Knight, "A Framework for Software Fault Tolerance in Real-Time Systems," *IEEE Transactions on Software Engineering* **SE-9**(3), pp. 355-364 (May 1983).

(3) T. Anderson, P. A. Lee, and S. K. Shrivastava, "A Model of Recoverability in Multilevel Systems," *IEEE Transactions on Software Engineering* **SE-4**(6), pp. 486-494 (November 1978).

(4) T. Anderson and P. A. Lee, "The Provision of Recoverable Interfaces," *Digest of Papers FTCS-9: Ninth Annual Symposium on Fault-Tolerant Computing*, pp. 87 (June 1979).

(5) T. Anderson and P. A. Lee, *Fault-Tolerance: Principles and Practice*, Prentice Hall International, London (1981).

(6) A. Avizienis, "Fault-Tolerant Systems," *IEEE Transactions on Computers* **C-25**(12), pp. 1304-1312 (December 1976).

(7) E. Best, "Atomicity of Activities," *Lecture Notes in Computer Science*, **Vol. 84**, ed. W. Brauer, Springer-Verlag, Berlin, pp. 225-250 (1980).

(8) R. H. Campbell, K. H. Horton, and G. G. Belford, "Simulations of a Fault-Tolerant Deadline Mechanism," *Digest of Papers FTCS-9: Ninth Annual Symposium on Fault-Tolerant Computing*, pp. 95-101 (1979).

(9) J. R. Connet, E. J. Pasternak, and B. D. Wagner, "Software Defences in Real-Time Control Systems," *Digest of Papers FTCS-2: Second Annual Symposium on Fault-Tolerant Computing*, pp. 94 (June 1972).

(10) C. T. Davies, "Recovery Semantics for a DB/DC System," *ACM 73 Annual Conference*, pp. 136 (August 1973).

(11) C. T. Davies, "Data Processing Spheres of Control," *IBM Systems Journal* **17**(2), pp. 179-198 (1978).

(12) S. T. Gregory and J. C. Knight, "A New Linguistic Approach to Backward Error Recovery," *Digest of Papers FTCS-15: Fifteenth International Conference on Fault-Tolerant Computing*, pp. 404-409 (1985).

(13) S. T. Gregory, *Programming Language Facilities for Backward Error Recovery in Real-Time Systems*, Ph.D. Dissertation, Department of Applied Mathematics and Computer Science, University of Virginia, Charlottesville, Virginia (1986).

(14) Per Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, NJ (1977).

(15) H. Hecht, "Fault Tolerant Software for Real-Time Applications," *ACM Computing Surveys* **8**(4), pp. 391-407 (December 1976).

(16) H. Hecht, "Fault-Tolerant Software," *IEEE Transactions on Reliability* **R-28**(3), pp. 227-232 (August 1979).

(17) C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM* **21**(8), pp. 666-677 (August 1978).

(18) J. J. Horning, et al, "A Program Structure for Error Detection and Recovery," *Lecture Notes in Computer Science*, **Vol. 16**, ed. E. Gelenbe and C. Kaiser, Springer-Verlag, Berlin, pp. 171-187 (1974).

(19) P. Jalote and R. H. Campbell, "Fault Tolerance Using Communicating Sequential Processes," *Digest of Papers FTCS-14: Fourteenth International Conference on Fault-Tolerant Computing*, pp. 347-352 (1984).

(20) P. Jalote and R. H. Campbell, "Atomic Actions for Fault-Tolerance Using CSP," *IEEE Transactions on Software Engineering* SE-12(1), pp. 59-68 (January 1986).

(21) K. H. Kim and C. V. Ramamoorthy, "Failure-Tolerant Parallel Programming and its Supporting System Architecture," *AFIPS Conference Proceedings 1976 NCC*, **Vol. 45**, pp. 413 (June 1976).

(22) K. H. Kim, "Strategies for Structured and Fault-Tolerant Design of Recovery Programs," *Proceedings COMPSAC 78*, pp. 651 (November 1978).

(23) K. H. Kim, "An Approach to Programmer-Transparent Coordination of Recovering Parallel Processes and its Efficient Implementation Rules," *Proceedings 1978 International Conference on Parallel Processing* (August 1978).

(24) K. H. Kim, "Approaches to Mechanization of the Conversation Scheme Based on Monitors," *IEEE Transactions on Software Engineering* SE-8(3), pp. 189-197 (May 1982).

C-2

(25) K. H. Kim, "Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults," *Proceedings: 4th Conference on Distributed Computing Systems*, pp. 526-532 (1984).

(26) H. Kopetz, "Software Redundancy in Real Time Systems," *IFIP Congress 74*, pp. 182-186 (August 1974).

(27) P. A. Lee, "A Reconsideration of the Recovery Block Scheme," *Computer Journal* **21**(4), pp. 306-310 (November 1978).

(28) Y-H. Lee and K. G. Shin, *Design and Evaluation of a Fault-Tolerant Multiprocessor Using Hardware Recovery Blocks*, University of Michigan Computing Research Laboratory Report CRL-TR-6-82 (August 1982).

(29) A. L. Liestman and R. H. Campbell, "A Fault-Tolerant Scheduling Problem," *Digest of Papers FTCS-13: Thirteenth Annual Symposium on Fault-Tolerant Computing*, pp. 42-47 (1983).

(30) D. B. Lomet, "Process Structuring, Synchronization and Recovery Using Atomic Actions," *ACM SIGPLAN Notices* **12**(3), pp. 128-137 (March 1977).

(31) L. MacLaren, "Evolving Toward Ada in Real Time Systems," *ACM SIGPLAN Notices* **15**(11), pp. 146-155 (November 1980).

(32) P. M. Melliar-Smith and B. Randell, "Software Reliability: the Role of Programmed Exception Handling," *ACM SIGPLAN Notices* **12**(3), pp. 95-100 (March 1977).

(33) P. M. Merlin and B. Randell, "State Restoration in Distributed Systems," *Digest of Papers FTCS-8: Eighth Annual Symposium on Fault-Tolerant Computing*, pp. 129-134 (1978).

(34) G. J. Myers, *Software Reliability: Principles and Practices*, Wiley, NY (1976).

(35) D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM* (December 1972).

(36) B. Randell, P. A. Lee, and P. C. Treleaven, *Reliable Computing Systems*, University of Newcastle upon Tyne Computing Laboratory Report 102 (May 1977).

(37) B. Randell, "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering* **SE-1**(2), pp. 220-232 (June 1975).

(38) D. L. Russel and M. J. Tiedeman, "Multiprocess Recovery Using Conversations," *Digest of Papers FTCS-9: Ninth Annual Symposium on Fault-Tolerant Computing*, pp. 106 (June 1979).

(39) D. L. Russel, "Process Backup in Producer-Consumer Systems," *ACM SIGOPS Operating Systems Review* **11**(5), pp. 151-157 (November 1977).

(40) D. L. Russel, "State Restoration in Systems of Communicating Processes," *IEEE Transactions Software Engineering* **SE-6**(2), pp. 183 (March 1980).

(41) K. G. Shin and Y-H. Lee, *Analysis of Backward Error Recovery for Concurrent Processes with Recovery Blocks*, University of Michigan Computing Research Laboratory Report CRL-TR-9-83 (February 1983).

(42) S. K. Shrivastava and J. P. Banatre, "Reliable Resource Allocation Between Unreliable Processes," *IEEE Transactions on Software Engineering* SE-4(3), pp. 230 (May 1978).

(43) S. K. Shrivastava, "Concurrent Pascal with Backward Error Recovery: Implementation," *Software-Practice and Experience* 9(12), pp. 1021-1033 (December 1979).

(44) S. K. Shrivastava, "Concurrent Pascal with Backward Error Recovery: Language Features and Examples," *Software-Practice and Experience* 9(12), pp. 1001-1020 (December 1979).

(45) S. K. Shrivastava, "Structuring Distributed Systems for Recoverability and Crash Resistance," *IEEE Transactions on Software Engineering* SE-7(4), pp. 436-447 (July 1981).

(46) R. M. Simpson, *A Study in the Design of Highly Integrated Systems*, University of Newcastle upon Tyne Computing Laboratory Report 67 (November 1974).

(47) J. S. M. Verhofstad, *On Multi-Level Recovery: An Approach Using Partially Recoverable Interfaces*, University of Newcastle upon Tyne Computing Laboratory Report 100 (May 1977).

(48) J. S. M. Verhofstad, *Recovery for Multi-Level Data Structures*, University of Newcastle upon Tyne Computing Laboratory Report 96 (December 1976).

(49) A. Y. Wei, K. Hiraishi, R. Cheng, and R. H. Campbell, "Application of the Fault-Tolerant Deadline Mechanism to a Satellite On-Board Computer System," *Digest of Papers FTCS-10: Tenth Annual Symposium on Fault-Tolerant Computing*, pp. 107-109 (1980).

(50) A. J. Wellings, D. Keeffe, and G. M. Tomlinson, "A Problem with Ada and Resource Allocation," *ACM Ada Letters* III(4), pp. 112-124 (January-February 1984).

(51) W. G. Wood, *Recovery Control of Communicating Processes in a Distributed System*, Computing Laboratory, University of Newcastle upon Tyne Report 158 (November 1980).

APPENDIX 5

REPORT LIST

REPORT LIST

The following is a list of papers and reports, other than progress reports, prepared under this grant.

(1) Knight, J.C. and J.I.A. Urquhart, "Fault-Tolerant Distributed Systems Using Ada", Proceedings of the *AIAA Computers in Aerospace Conference*, October 1983, Hartford, CT.

(2) Knight, J.C. and J.I.A. Urquhart, "The Implementation And Use Of Ada On Fault-Tolerant Distributed Systems", *Ada LETTERS*, Vol. 4 No. 3 November 1984.

(3) Knight, J.C. and J.I.A. Urquhart, "On The Implementation and Use of Ada on Fault-Tolerant Distributed Systems", *IEEE Transactions on Software Engineering*. to appear.

(4) Knight J.C. and S.T. Gregory, "A Testbed for Evaluating Fault-Tolerant Distributed Systems", Digest of Papers FTCS-14: *Fourteenth Annual Symposium on Fault-Tolerant Computing*, June 1984, Orlando, FL.

(5) Knight J.C. and S.T. Gregory, "A New Linguistic Approach To Backward Error Recovery", Digest of Papers FTCS-15: *Fifteenth Annual Symposium on Fault-Tolerant Computing*, June 1985, Ann Arbor, MI.

(6) Gregory, S.T. and J.C. Knight, "Concurrent System Recovery" in *Resilient Computing Systems, Volume 2* edited by T. Anderson, Wiley, 1987.

(7) Knight, J.C. and M.E. Rouleau, "Analysis Of Ada For A Crucial Distributed Application", Proceedings of the Fifth National Conference On Ada Technology, Washington DC, March, 1987.

(8) Knight, J.C. and J.I.A. Urquhart, "Difficulties With Ada As A Language For Reliable Distributed Processing", Unpublished.

(9) Knight, J.C. and J.I.A. Urquhart, "Programming Language Requirements For Distributed Real-Time Systems Which Tolerate Processor Failure", Unpublished.

DISTRIBUTION LIST

Copy No.	
1 - 3	National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665 Attention: Mr. Michael Holloway ISD, MS 478
4 - 5	NASA Scientific and Technical Information Facility P. O. Box 8757 Baltimore/Washington International Airport Baltimore, MD 21240
6 - 7	J. C. Knight, CS
8	R. P. Cook, CS
9 - 10	E. H. Pancake, Clark Hall
11	SEAS Publication Files

JO#8729:rsr