DAA/LANGLEY

Semi-Annual Progress 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. Edmond H. Senn
ISD M/S 125

Submitted by:

J. C. Knight
Associate Professor

Report No. UVA/528213/CS87/109
August 1986

SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA 22901

Semi-Annual Progress 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. Edmond H. Senn
ISD M/S 125

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/CS87/109 Copy No. _____
August 1986

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.

The reduced cost of computer hardware and the expected advantages of distributed processing (for example, increased reliability through redundancy and greater flexibility) indicate that many aerospace computer systems will be distributed. The use of Ada and distributed systems seems like a good combination for advanced aerospace embedded systems.

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

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

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) The preparation 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) 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 incomplete and subject to change. Appendix 3 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. Section and figure numbers as they will appear in the text have been retained in that appendix. A list of papers and reports prepared under this grant, other than the annual and semi-annual progress reports, is presented in Appendix 4.

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.

[Describe general efficiency measurement techniques which we will use.] [Since many of the computational details of our application are being neglected, how do we measure percentages accurately? Cpu overhead will be particularly difficult to measure.] 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 *may* 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

ATOPS SYSTEM REQUIREMENTS

This document is the high-level software requirements specification for an aircraft computing system capable of performing all flight tasks from take-off through landing. The realization of this specification will not be adequate for any real system. We intend the structure of the desired program to be identical to that of a practical aircraft computing system, but 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.

System Overview

The computing facility on which this program will run should be viewed as a general distributed system comprising M loosely coupled processors and N memories where M and N exceed zero. Processors and peripherals will communicate via a global bus.

The computing system will access a number of peripheral devices. Flight sensors will provide situational information as input to computations which drive the effectors and provide corresponding situational data to the cockpit. Devices accessed by the pilot select the format and general content of the situational information provided to the cockpit displays, and an additional control panel allows the pilot to access or modify the current flight plan. Still another control panel allows the pilot to select the aircraft flight mode. A number of modes requiring varying degrees of direct pilot control of the aircraft are available.

System Data Flow

The high-level system data flow is depicted in figure 1.1. This section is devoted to descriptions of the data transformations represented by the boxes in the figure.

Positional Indicator Control Panels

The positional indicator control panels inform the positional indication computation module of the format and general content of the information to be represented on the positional indicator displays.

Path Control Panel

The path control panel provides the path definition module with three different kinds of information. Initialization data inform the system of the ground-level barometric pressure, starting latitude and longitude, and the time, among other things. Flight plan specifications determine the aircraft flight plan. Holding patterns and offset paths provide the path definition module with parameters necessary for configuration of the situational displays.

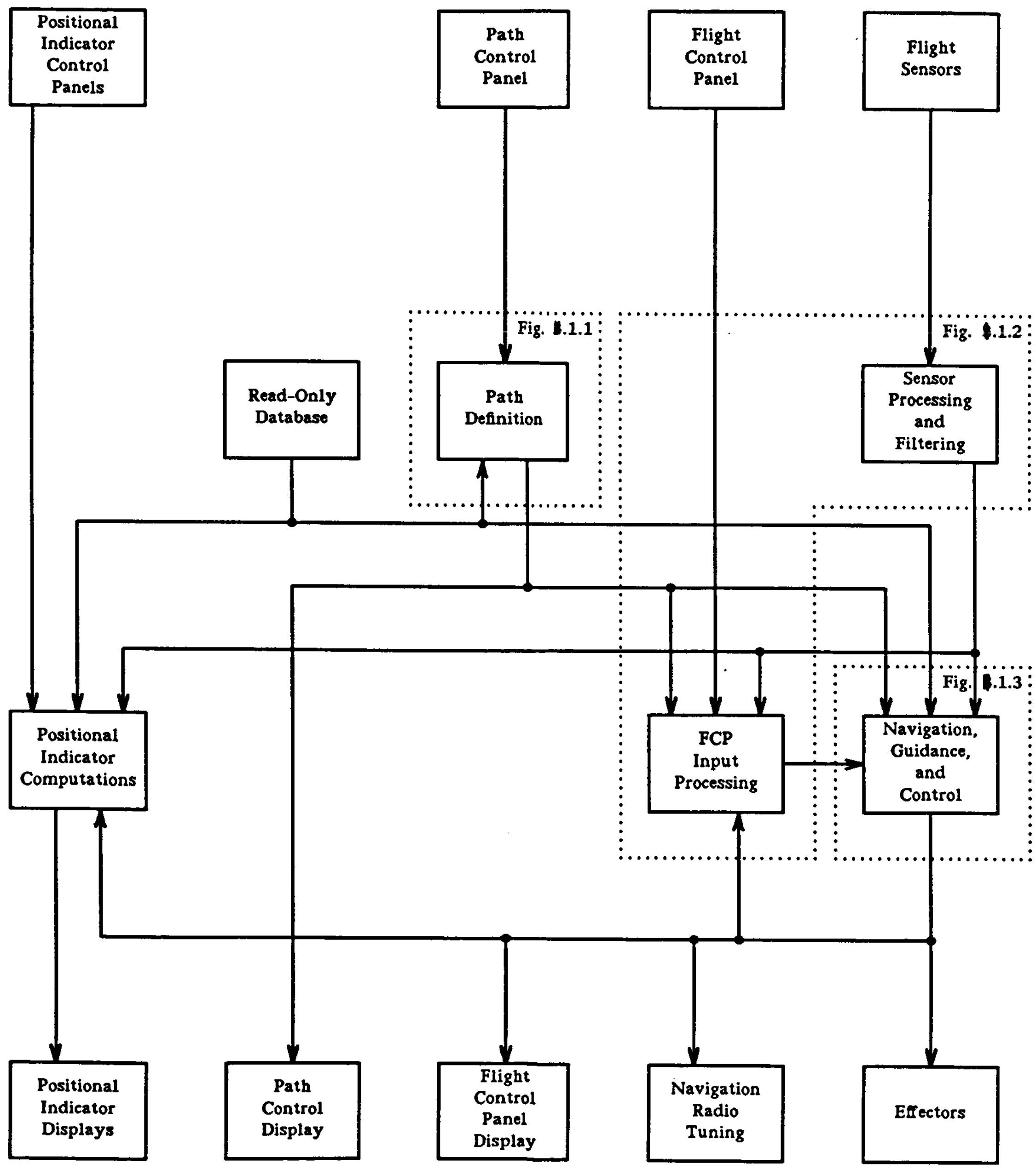

Positional Indicator Control Panels
Path Control Panel
Flight Control Panel
Flight Sensors
Fig. 3.1.1
Fig. 3.1.2
Read-Only Database
Path Definition
Sensor Processing and Filtering
Fig. 3.1.3
Positional Indicator Computations
FCP Input Processing
Navigation, Guidance, and Control
Positional Indicator Displays
Path Control Display
Flight Control Panel Display
Navigation Radio Tuning
Effectors

Fig. 1.1. System Data Flow

Flight Control Panel

The flight control panel provides the remainder of the system with facilities for the specification of velocity, altitude, and direction, for the selection of one of many flight modes ranging from direct pilot control to completely automatic operation, and for the activation of a variety of navigational sensor systems.

Flight Sensors

The flight sensors provide the computing system with navigational data, the operational statuses of various mechanical systems, and inputs from the brolly handles used to steer the aircraft in certain flight modes.

Read-Only Database

The system database contains information about terrain, airfields, ground-based navigational aids, routes, etc., which is used for guidance, path definition, and situational display.

Path Definition

The path definition module accepts instructions from the path control panel. It provides general data from the read-only database and specifics regarding the flight plan to the path control display for perusal by the pilot. It directs the positional indicator computation module to configure the positional indicators for display of pilot-specified holding patterns and offset

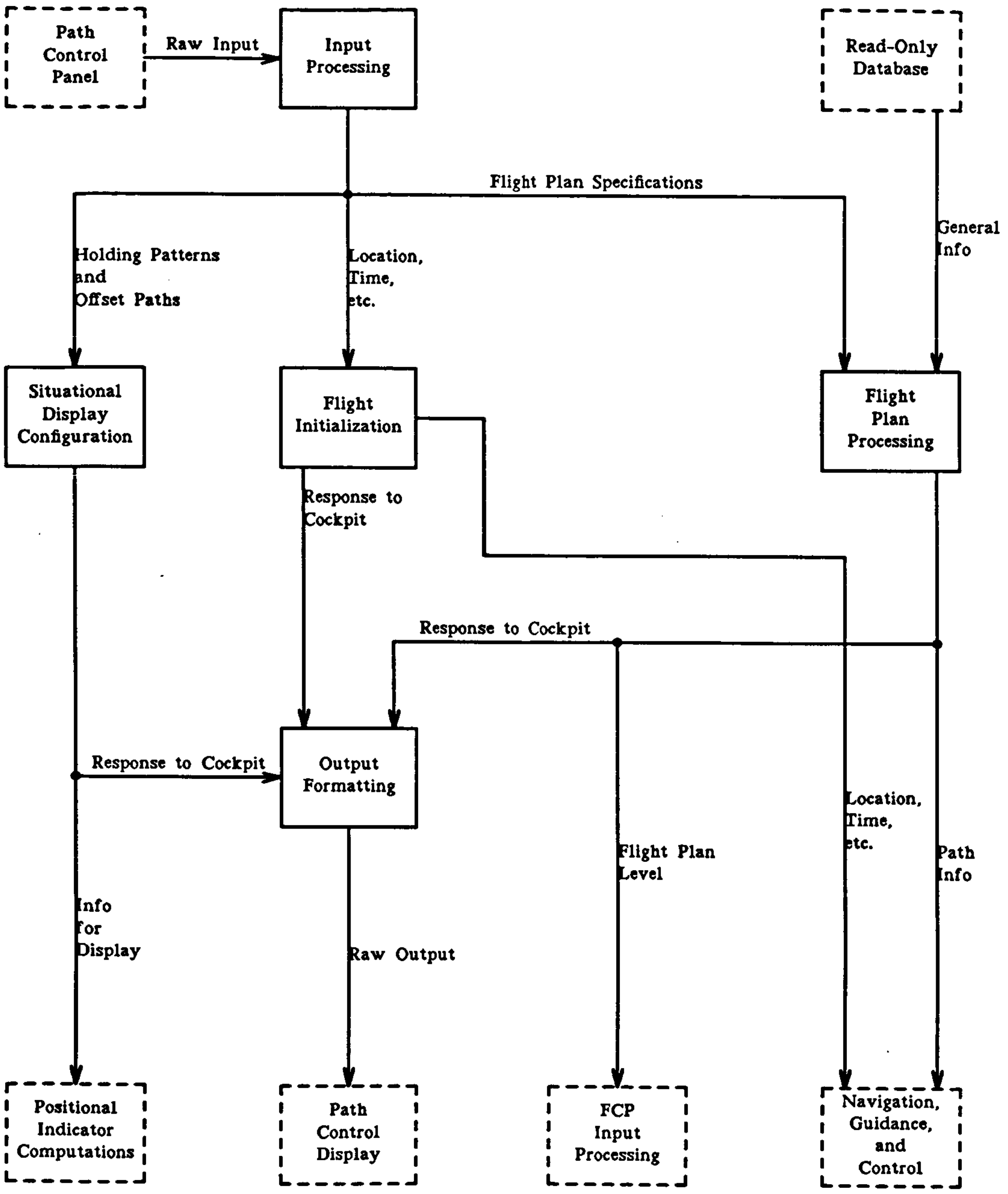


Fig. 1.1.1. Path Definition Data Flow

paths. It informs the flight control panel input processing module of the current flight plan level. It provides the particulars of the flight plan to the navigation, guidance, and control module. Details of the data flow associated with path definition are shown in figure 1.1.1.

Flight Controls and Flight Sensors Input Processing

Details of the data flow associated with flight control and sensor processing are shown in figure 1.1.2.

Navigation, Guidance, and Control

Details of the data flow associated with navigation, guidance, and control are shown in figure 1.1.3.

Navigation

A further refinement of the details of the data flow associated with navigation is shown in figure 1.1.3.1.

Microwave Landing System

The Microwave Landing System (MLS) software uses the elevation, azimuth, and range inputs from the MLS hardware and the acceleration inputs from the body-mounted accelerometers to determine accurately position, velocity, and acceleration of the aircraft. MLS navigation information is accurate enough to support fully automatic landing of the aircraft. MLS is a component of the navigation module and a further

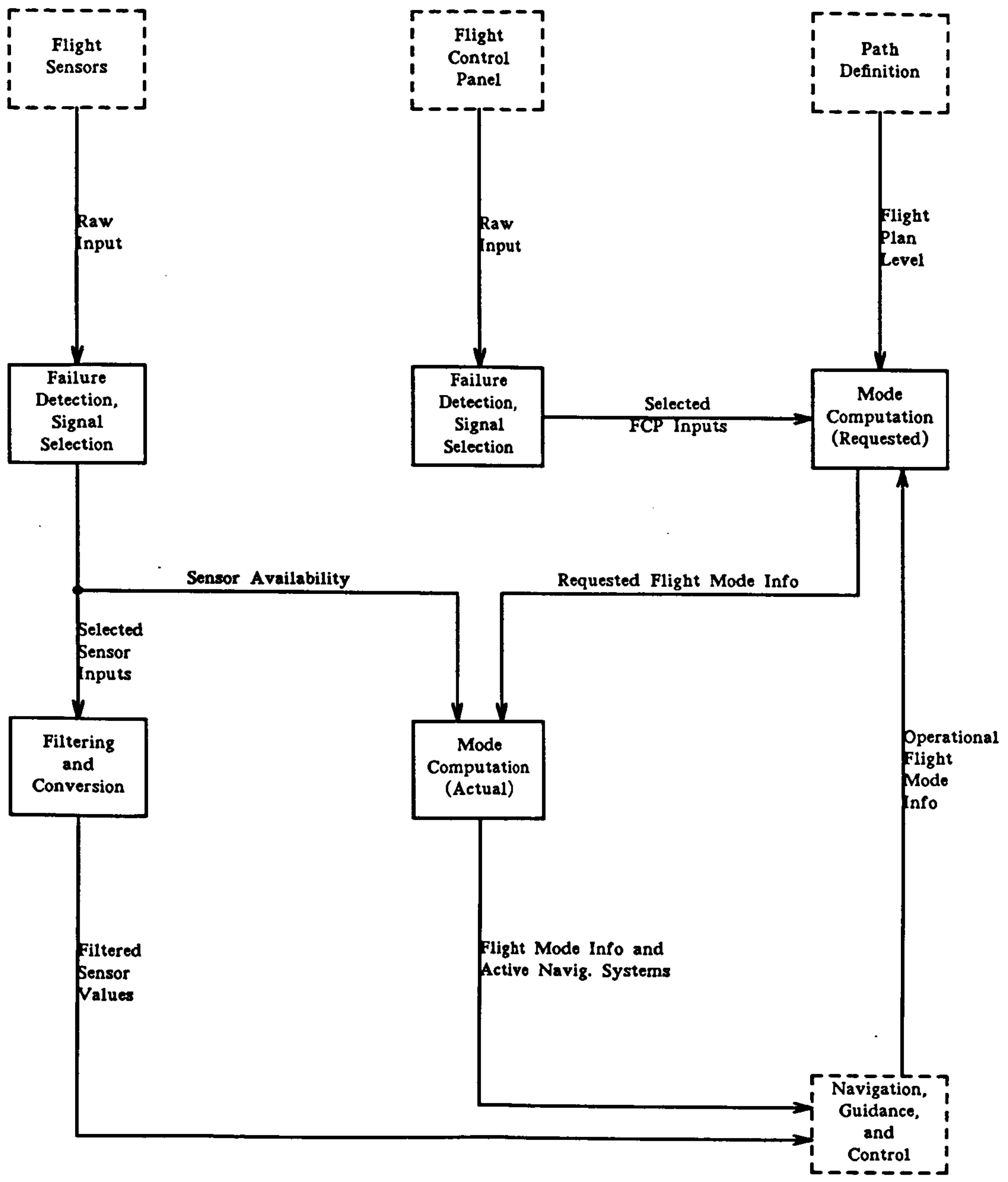

Flight Sensors
Flight Control Panel
Path Definition
Raw Input
Raw Input
Flight Plan Level
Failure Detection, Signal Selection
Failure Detection, Signal Selection
Selected FCP Inputs
Mode Computation (Requested)
Sensor Availability
Requested Flight Mode Info
Selected Sensor Inputs
Filtering and Conversion
Mode Computation (Actual)
Operational Flight Mode Info
Filtered Sensor Values
Flight Mode Info and Active Navig. Systems
Navigation, Guidance, and Control

Fig. 1.1.2 Flight Controls and Flight Sensors Input Processing Data Flow

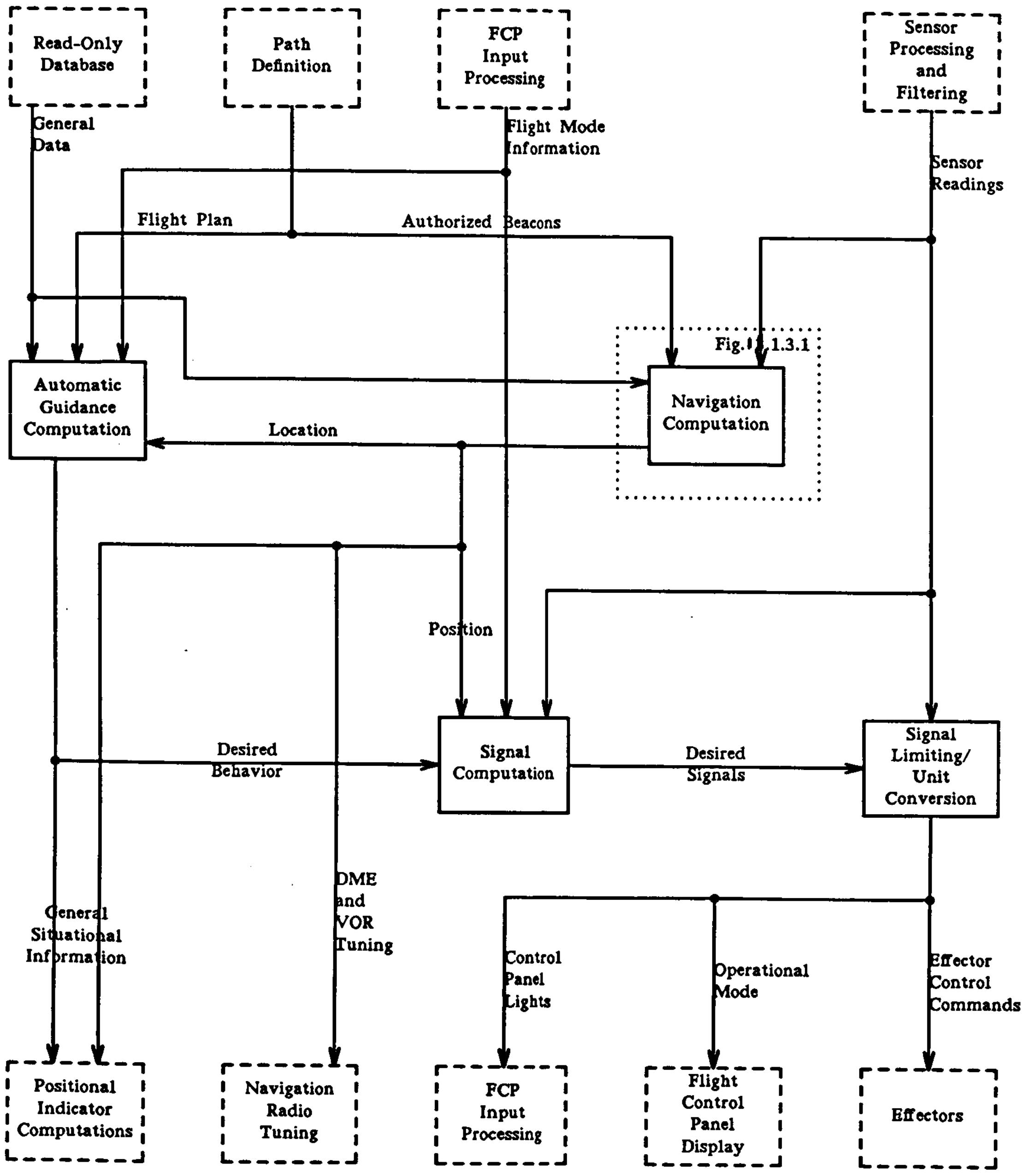
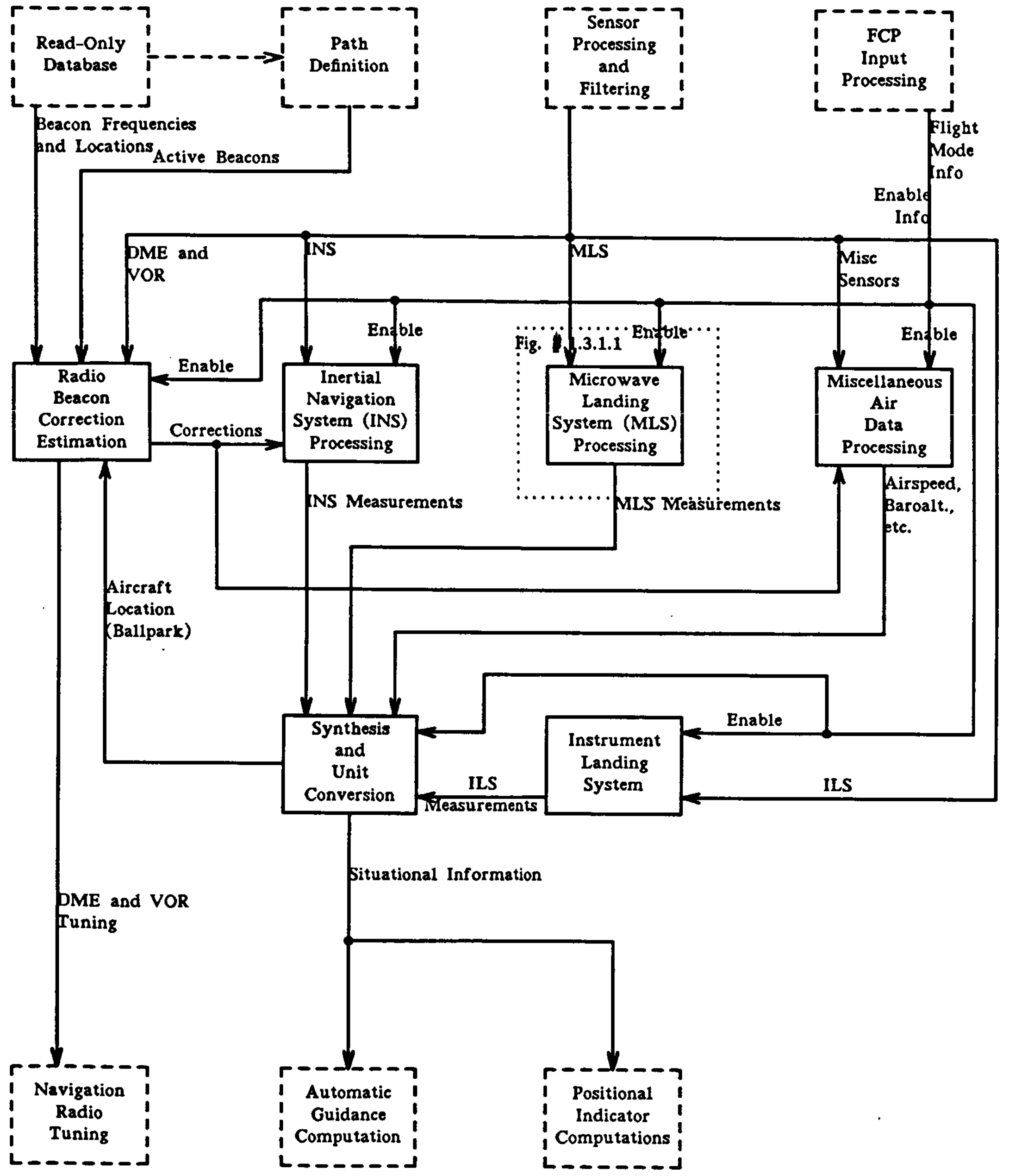


Fig. 1.1.3 Navigation, Guidance, and Control Data Flow



Read-Only Database
Path Definition
Sensor Processing and Filtering
FCP Input Processing
Beacon Frequencies and Locations
Active Beacons
Flight Mode Info
Enable Info
DME and VOR
INS
MLS
Misc Sensors
Enable
Enable
Fig. 1.1.3.1
Enable
Enable
Radio Beacon Correction Estimation
Inertial Navigation System (INS) Processing
Microwave Landing System (MLS) Processing
Miscellaneous Air Data Processing
Corrections
INS Measurements
MLS Measurements
Airspeed, Baroalt., etc.
Aircraft Location (Ballpark)
Synthesis and Unit Conversion
Instrument Landing System
Enable
ILS Measurements
ILS
Situational Information
DME and VOR Tuning
Navigation Radio Tuning
Automatic Guidance Computation
Positional Indicator Computations

Fig. 1.1.3.1 Navigation Data Flow

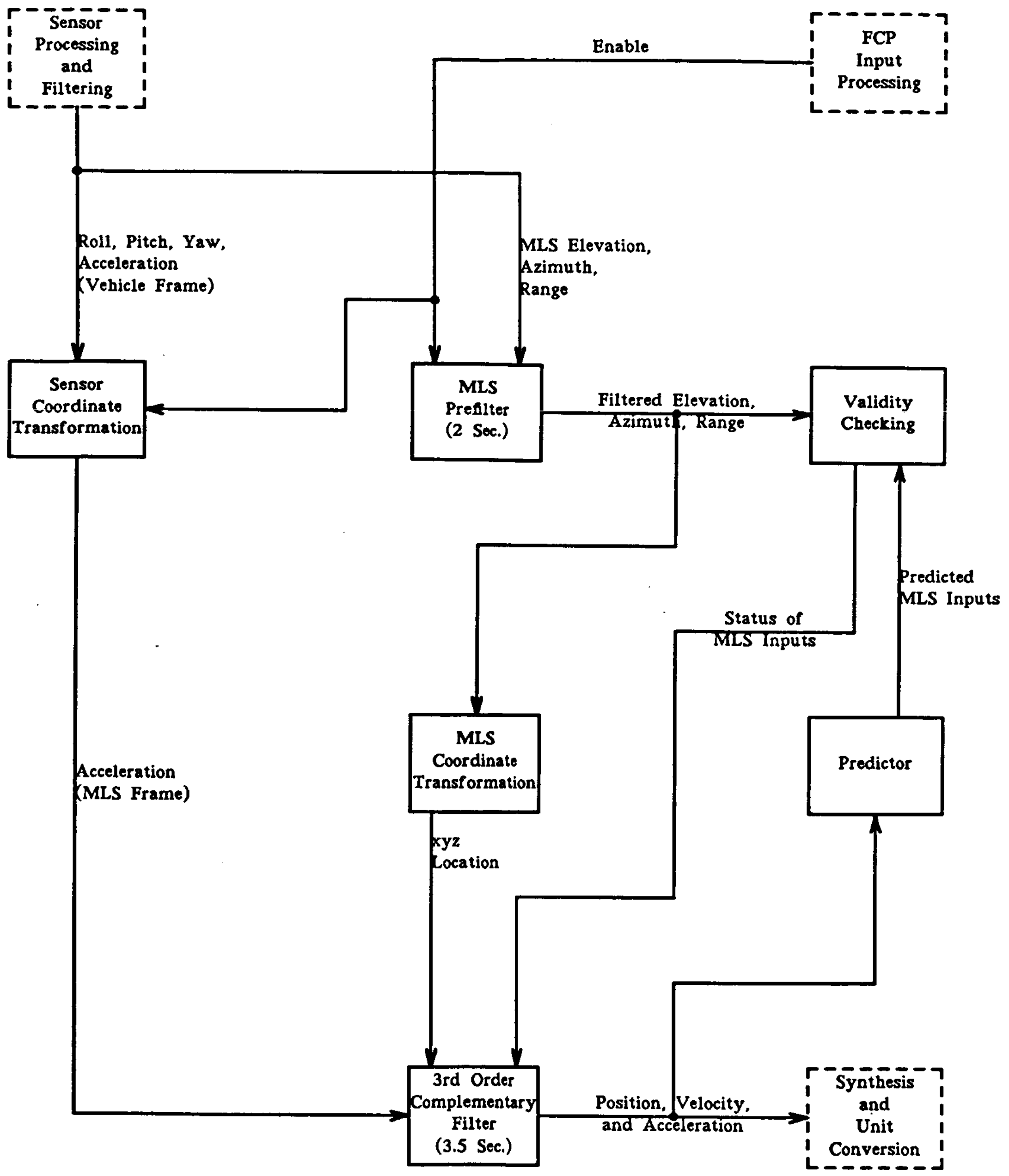

Sensor Processing and Filtering
Enable
FCP Input Processing
Roll, Pitch, Yaw, Acceleration (Vehicle Frame)
MLS Elevation, Azimuth, Range
Sensor Coordinate Transformation
MLS Prefilter (2 Sec.)
Filtered Elevation, Azimuth, Range
Validity Checking
Predicted MLS Inputs
Status of MLS Inputs
MLS Coordinate Transformation
Predictor
Acceleration (MLS Frame)
xyz Location
3rd Order Complementary Filter (3.5 Sec.)
Position, Velocity, and Acceleration
Synthesis and Unit Conversion

Fig. 1.1.3.1.1 Microwave Landing System (MLS) Data Flow

refinement of the data flow associated with MLS is shown in figure 1.1.3.1.1.

Positional Indicator Displays

The positional indicator displays provide the pilot with attitudinal and horizontal situation information in a number of formats.

Path Control Display

The path control display is the screen, similar to that of a computer terminal, on which the current path, a new path being input, or general information from the system database can be represented.

Sensor Feedback and Tuning

The sensor feedback and tuning module provides two kinds of feedback. The flight control panel display conveys information to the pilot regarding his efforts to control the aircraft. For example, if he selects a mode which, due to failure of a sensor, cannot be activated, the lighting configuration on the panel will alert him to the limitation. The radio tuner sets the navigation beacon receiver to the frequency of the desired beacon.

Effectors

The aircraft effectors control the mechanical functions of the aircraft. The ailerons, delta elevators, rudder, and throttle are all linked to the computing system via effectors.

APPENDIX 3

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

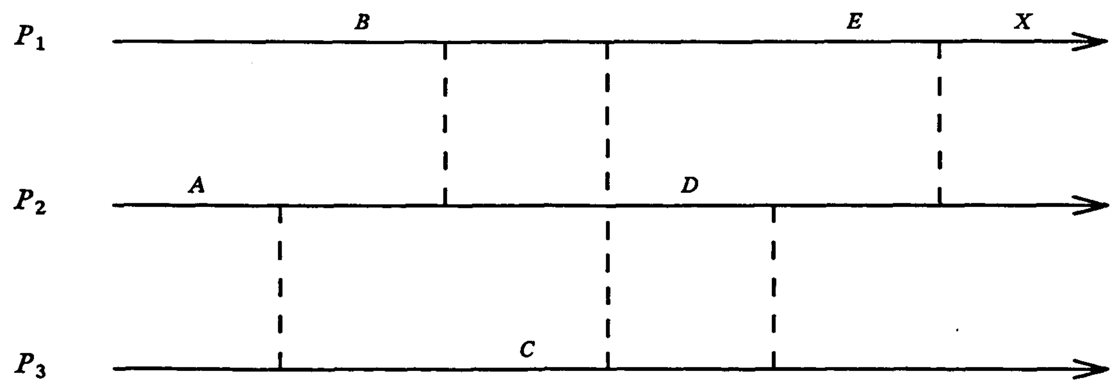
Recovery blocks provide a mechanism for building backward error recovery into sequential programs. However, many programs such as operating systems and real-time control systems, are *concurrent*. A concurrent system consists of a set of communicating sequential processes. The processes execute in parallel 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 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.



The Domino Effect

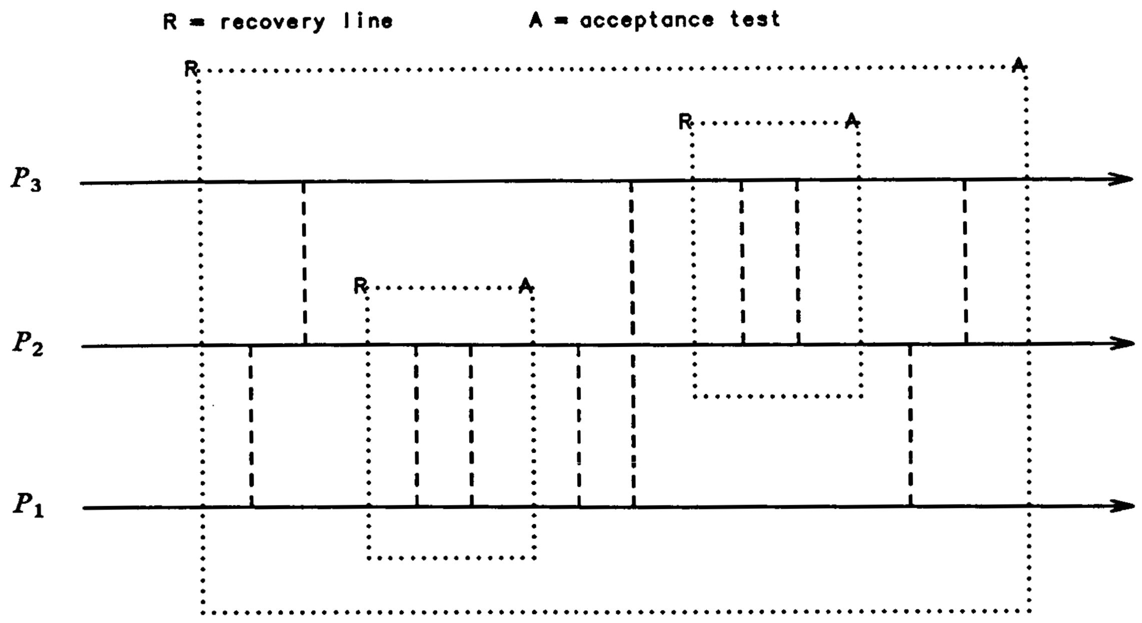
Figure 9.1

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



Three Nested Conversations

Figure 9.2

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 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*, 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.

```
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

```
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.

None of Russell's or Kim's conversation schemes enforce the prohibition against smuggling. If processes use monitors, message buffers, or ordinary

```
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

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.

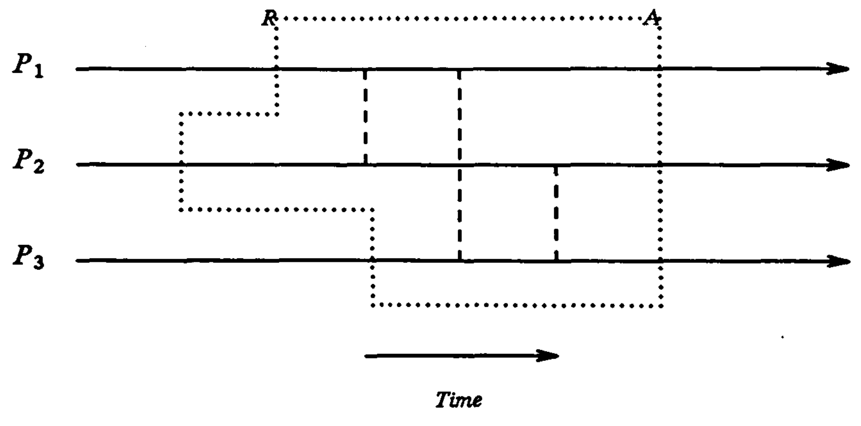
9.8.1. Dialogs

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*



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

associates a particular discuss statement 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.

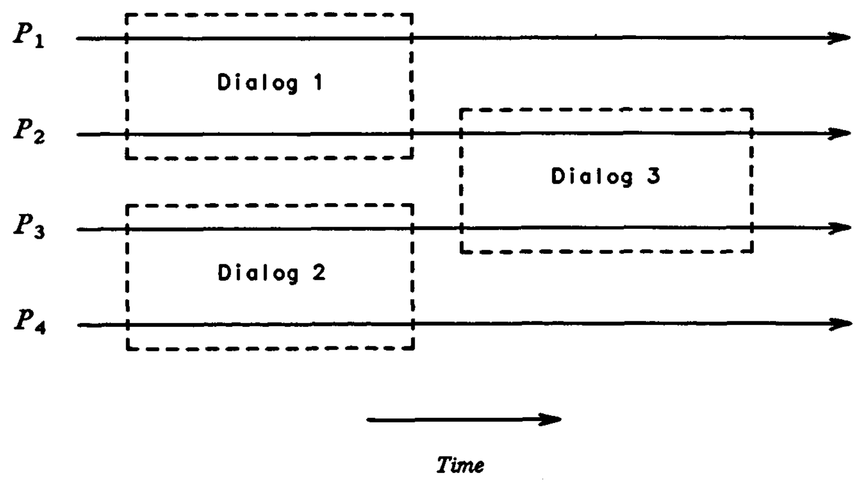
For the actions of the dialog's participants to appear atomic to other processes, all forms of communication must be controlled. The set of variables shared by processes participating in a dialog are locked by the compiler and execution-time support system to prevent smuggling. While locked, the shared 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.



Four Processes in a Colloquy of Three Dialogs

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.*

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

```
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

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 languagee. 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).

(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 4

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 J.I.A. Urquhart, "Difficulties With Ada As A Language For Reliable Distributed Processing", Unpublished.

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