PROGRAMMING LANGUAGE FACILITIES FOR BACKWARD ERROR RECOVERY IN REAL-TIME SYSTEMS

Samuel Thurston Gregory

Computer Science Report No. RM-86-08
January 1987

Programming Language Facilities for Backward Error Recovery in Real-Time Systems

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Samuel Thurston Gregory

January 1987

ABSTRACT

Programming Language Facilities

for

Backward Error Recovery

in

Real-Time Systems

Digital computers are being used increasingly in dedicated control applications that require high reliability. These applications are often embedded, concurrent, and operate in real time. Strategies are employed in hardware and software to tolerate hardware faults. Some work has also been done on methods of enabling the applications software to tolerate its faults. This dissertation researches the latter area, and focuses on mapping the ideal world of the theoretical software fault-tolerance proposals onto the real world of programs as influenced by their languages of expression, thus making software fault-tolerance proposals more amenable to use in real programs.

Issues involved in language facilities for backward error recovery in critical, real-time systems are examined. The previous proposals are found lacking. The *dialog*, a new building block for concurrent programs, and the *colloquy*, a new backward-error-recovery organization, are introduced to remedy the situation. The colloquy is a general linguistic approach to backward error recovery. It solves the problems raised with all previously proposed language structures for backward error recovery in both sequential and concurrent programs. Implementable programming language facilities are presented to embody the colloquy. The previous proposals are shown to be special cases of the colloquy. Thus, the colloquy provides a general framework for describing backward error recovery in concurrent programs.

We next examine the problem of providing backward error recovery in *realistic* programming languages of which Ada is an example. This examination reveals several new problems that have not

been addressed previously. We show that the source of the problems is the continuous need to be able to define a recovery line so as to be able to perform state restoration. Many language facilities that have not been addressed by other researchers, such as shared data, process creation and destruction, and pointers, make the establishment of a recovery line extremely difficult. Facilities designed to encourage use of certain programming methods also contribute to this difficulty. We present solutions to some of the problems identified.

The fragility of these solutions and absence of solutions to many of the problems bespeaks the need to include backward error recovery as a basic design goal of a programming language rather than merging it into an existing language.

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Computer Science)

[illegible]
AUTHOR

This dissertation has been read and approved by the Examining Committee:

[illegible]
Dissertation Advisor

[illegible]
Committee Chairman

[illegible]

[illegible]

[illegible]

Accepted for the School of Engineering and Applied Science:

Dean, School of Engineering and Applied Science

January 1987

Dedicated To The Memory Of My Mother

Jean Marie Clark Gregory

August 19, 1923 - April 22, 1962

Acknowledgements

I would like to thank my advisor, John Knight, for the encouragement, guidance and assistance he provided during the development and writing of this dissertation. He insisted that ideas which I originally considered insignificant were substantial and vice versa; he continued to push me to finish this dissertation each of the many times I became stalled.

I would also like to thank the other members of my committee; Alan P. Batson, Paul F. Reynolds Jr., Jack W. Davidson, and Edward A. Parrish for their advice and assistance.

The referees for FTCS-15 and FTCS-16 showed much interest in these results. Tom Anderson also offered encouragement by expressing interest in this work.

Additional thanks to the Department of Computer Science for providing computing resources for this research. Specifically, Mark Smith supplied the disk space and kept the printer working. NASA paid for my food, shelter and tuition for most of my studies in exchange for work under grants to my advisor.

Alex Colvin, Andy Lacy, Dave Stotts, and many other graduate students, especially the ''lunch group'', listened to me when I needed to talk about this work or when I just needed a break. Will Omohundro became the best friend I have ever had and, along with Allen Hemberger, helped me maintain my sanity through outside activities and interests.

Table of Contents

List of Figures

CHAPTER 1

Introduction

There are applications of computers that require a very high degree of reliability. Many such applications operate in real time and are often embedded. A real-time, embedded system is termed *crucial* if a failure to provide its service at an acceptable level could endanger human life or equipment under the system's control. Examples of crucial systems include control systems for aircraft, nuclear power plants, and spacecraft.

The reliability required of crucial systems often entails the provision of service over long periods without human intervention. This might be achieved via the combination of two approaches. First, the systems are designed and built using every available means of *fault avoidance* followed by *fault removal* [Ande81], so as to eliminate as many faults as possible. This approach is practiced often for the software components of these systems. The second step is to prepare the system to tolerate those faults which will assuredly escape detection during the fault avoidance activity. This approach, called *fault tolerance*, is being employed only minimally and haphazardly in the software components of crucial systems. This dissertation examines the consistent, organized use of one approach to fault tolerance in the software components of crucial systems.

The traditional methods of tolerating faults in general all employ some kind of redundancy. Multiple instances of parts of the system (or of the system itself) are included. When a fault is detected in one instance, that instance is replaced by another in the hope that the new instance will not contain the same fault. This general method influences the design of the software.

Software does not wear out or otherwise deteriorate, nor in general do accidents affect it. Software faults are exclusively design faults, and as such are not treatable by simple replication of the software.

The overall problem, then, is one of designing embedded real-time software systems in a manner that will allow them to survive and tolerate faults in that design.

On the face of it, the idea of designing software to tolerate faults in its design seems self-contradictory. It is not. In the remainder of this chapter, we develop the problem's context. We first survey sample applications for which the problem needs to be solved. Next, some of the consequences of not solving the problem are outlined. Some assumptions are made concerning the targeted systems and about very low-level software support. The existing notation of the research community in fault tolerance is then reviewed before a detailed description of the problem is provided with considerations motivating a search for its solution. Finally, we outline the rest of the dissertation.

1.1. Applications Providing Context

An *embedded system* is a computer system that is part of a larger system that it controls in real time. Certain applications of embedded systems that give the context of our problem are described briefly below.

1.1.1. Aircraft Control

In some present [Meye81] and many future [Groo82] aircraft control systems, a computer system takes advice from the pilot through his actions. The joystick, pedals, and throttle are connected to transducers being read by the system rather than to pressurized hydraulic lines. The pilot is totally dependent upon the computer system for correct operation of the vehicle. The replacement of redundant hydraulic control lines by electrical wires or fiber-optic cable will have greatly reduced the overall weight already. In extreme cases, the airframe will have been weakened for weight reduction, hence fuel savings. Lighter materials being considered, such as composites, and thinner sheets and structural members of the traditional metals leave an airframe too flexible for stability under the enormous stresses of flight [Ande81]. The computer system will be responsible for compensating for this increased flexibility by sensing mounting stresses and adjusting small control panels to re-adjust air flows over the airframe.

1.1.2. Nuclear Power Plant Control

Although current regulations prohibit full-authority, computerized control of nuclear power plants in the United States, such control is being evaluated here and is being employed elsewhere. Control of nuclear power plants would present a situation with similarities to that of aircraft [Long77]. In other countries [Neum81], regular control has been transferred to the computer system with the operators giving advice and acting as a check on the computer system. The programmed control *must* be on time and correct or ready to and capable of backing the power plant into a relatively safer state, i.e. provide a degraded yet acceptable service.

1.1.3. Spacecraft Control

Spacecraft control is similar in many respects to aircraft control. An example of an existing system is the NASA Space Shuttle. "The Space Shuttle is a vehicle which is *totally* dependent on the proper operation of its computers; a mission cannot even be aborted if the computers fail." ([Ande81] p.2) "Note that landing cannot be achieved manually." ([Neum84] p.4) On manned spacecraft, the computer system is responsible for controlling and maintaining the environment which supports human lives as well as controlling the craft itself [Ande83a]. Faults on unmanned craft might cause direct damage to the craft, cause it to go off course, or cause it to misalign its antennae so as to prevent future human intervention [Eber83].

1.2. Consequences of Failure

If faults manifest themselves and alternate actions are not provided and imposed, each of these applications could experience failures which would result in deaths or injuries. Other results could be damage to or loss of equipment of great value. Value, here, can be in terms either of money or of time and effort required to replace or reinstall the equipment.

Failure of the controlling software in an aircraft severs what little influence the pilot had. Further, in the case of an airframe employing active controls, true manual controls would be useless to a pilot since the air surfaces must be adjusted at rates far faster than human reaction times. We have already heard of several instances in which failed hardware devices have brought the world close to war [Neum80]. Failing software might not be so easy to detect externally. A failing system might try to launch missiles without human direction or refuse to launch when so directed.

1.2.1. Actual Failures Due to Computer System Faults

Many of the major problems in the Apollo series of space missions were direct results of software design faults [Bohe73, Ulsa73]. An invalid sequence of actions by the astronauts on board Apollo 8 was allowed to destroy data in the computer's erasable memory. A data overload from the rendezvous radar during the last seconds of the Apollo 11 lunar landing resulted in the need for a blind, manual landing [Bens84]. Eighteen (unspecified) bugs were found during the ten-day flight of Apollo 14. In 1971, a French satellite, rather than interrogating over one hundred weather balloons for their data, sent destruct commands to seventy-two of them. An early U.S. Mariner mission's loss is also attributed to software faults. The first launch of the NASA Space Shuttle was delayed for two days due to a software fault [Garm81]. In that case, the software system's recovery action of disabling further mission progress was valid. However, had the fault appeared after launch, the same response would have been catastrophic. One Space Shuttle fault, which was fortunately caught during a simulation exercise, would have locked the software into a tight loop had the astronauts been required to abort a mission after a prior canceled abort [Neum82]. There is a case of an attempted missile firing from the wing of a jet fighter in which the software re-closed the supporting clamp before the missile could get clear [Neum83]. A nuclear reactor vented radioactive material into the atmosphere after a false alarm had caused the controlling software to stop all mechanical motion within the reactor at a time when that policy was inapplicable [Neum83]. Myers gives a further list of failures due to software faults [Myer76].

1.3. Assumptions

1.3.1. Hardware Fault Tolerance

The people proposing and building crucial systems recognize that the hardware comprising them will tend to wear out or will be subject to accident or deliberate damage during the systems' useful lifetimes. To protect against this, multiple instances of devices are employed. To guard against similar failure modes such as might appear due to hardware design faults or design weaknesses, the different computers may even be of different manufacture. Anderson and Lee [Ande81] give some reasons tolerance of hardware design faults is not always provided in practice.

Our focus is on software tolerating applications software faults. It is assumed that measures have been taken to mask any hardware faults from the application software [Corn82, Corn83]. One of the issues arising from this assumption is the possibility of untoward effects of a processor failure anywhere in the system during the handling of a software fault. The possibility of hardware tolerating software faults seems rather meaningless. The field of architectures for reliability seems largely concerned with fault detection with little regard for responses allowing continued service.

1.3.2. Language Support

We realize that the programming language implementation can be a source of faults in the overall software system. Assuring the correct operation of this form of large and complex system is a recognized and open problem. However, in order to say anything meaningful we must assume that failures from this source will not befall us. To an application software system, a faulty language support system would be essentially indistinguishable from a processor that incorrectly interprets instructions. "Because of the powerful nature of interpretive interfaces it cannot be hoped to protect a process from the underlying interpreter." ([Ande81] p.164) We must assume that fault-tolerance measures are taken within that system so that we are provided with a fault-tolerant interface.

1.4. Fault-Tolerance Terminology

The duplication of code units and the reconfiguration duties of the software in response to different forms of failures is generally called *software fault tolerance*. Strategies (in most cases redundancy) are adopted to enable the system to *tolerate* or survive, with perhaps lessened or degraded capabilities for providing the intended service, *faults* that, for whatever reasons, develop in the component *software*.

Up until this point, we have been using terms like "fault" without definition, relying upon colloquial or intuitive understanding. The study of fault tolerance has produced a certain terminology which will be useful in further discussion of this subject.

> A *system* is a set of interacting components and a design that prescribes and controls the pattern of interaction; as a unit in larger context, a system is a component that maintains a pattern of behavior at an interface between it and its environment [Ande81].
>
> An *interface* is simply a place of interaction between two systems [Ande81].

At the lowest level, a software system's components are the hardware instructions and the design is the arrangement of their use. Since we assume hardware fault tolerance, all faults will be found in the designs at various levels of systems.

> A *measure* is a construction within the design of a system intended to perform a specific task [Ande81].

A measure deals with the interaction of components of a system, and only indirectly affects its external behavior.

> A *mechanism* is a construction within an interpreter that provides a specific facility available on the interpretive interface [Ande81].

When a system is viewed as a black box, its behavior is analogous to an interpreter or a stimulus-response machine. Some subset of the stimulus-response pairs are often due to a small group of the system's components and part of its design. Such a group of components and part of the design together qualify as a mechanism. Thus, a mechanism directly affects a system's external behavior. A mechanism may be constructed intentionally or by happenstance; whereas a measure is only intentional.

> A *failure* is a deviation of a system's external behavior from its specified behavior [Ande83b].

There is an assumption here, as in discussions on testing and verification, that there *is* a formal, detailed, and complete specification of what the system's external behavior should be. A consequence of this definition and that of a system, is that a failure can occur in a subsystem or component and be tolerated by a larger system without bringing about a failure of the larger system.

> An *error* is a state of a system that could lead to a failure if not corrected [Ande83b].

An error might not actually lead to a failure. Pairs of ''bugs'' that have the effect of compensating for each other in certain circumstances are well known in the computing folklore. In the case of mutually-correcting pairs of ''bugs'', the error(s) might never be detected, but exist nonetheless.

Tolerating a fault (see below) is the process of correcting a system's state so that it does not lead to a failure. By correcting a system's state, we do not mean correcting the system. Rather, we mean bringing the system's set of variable values (including current point(s) of control) into a configuration that is consistent and unlikely to lead directly into another erroneous configuration. The repaired state may be an actual former state or a newly-constructed one.

> A *fault* is the adjudged cause of an error [Ande83b]. This can be thought of as a defective value in the state of a component or in the design of a system [Ande81].

A fault in a system may be the failure of one of its components. Colloquially, a failure is an execution-time thing while a design fault actually exists long before execution-time. The expanded definition allows us to take advantage of the multitude of levels of systems and subsystems so that if a fault is not tolerated at one level, it might be tolerated at a higher level. We are using the word "design" in a broader sense than is usual. A mistake in coding or implementing software is considered a design fault.

Tolerating faults is a process that involves certain activities. These are: *error detection*, *damage assessment*, *error recovery*, and *continued service* [Ande83b].

Error Detection

Error detection involves checking values in the system's state for consistency and reasonableness. This includes hardware checks as in the canonical example of a division by zero. In software, acceptability checks are placed within the code. If the checks are on inputs, they are called defensive checks. Other checks occurring after an activity detect erroneous results before they are passed on. Another kind of check is the use of a timer. It may be specified that, if an activity takes more than a reasonable length of time, that activity should be terminated and an error signaled. In a reasonably complicated system, checks for acceptability, consistency, and reasonableness would test more high-level relationships between values in the state than checks on single variable values. Checks for specific malfunctions are considered checks for *anticipated faults*. There is some argument that, if a software design fault can be anticipated, it should be eliminated during design rather than checked for and tolerated under service [Mell77].

Software fault-tolerance research aims at *unanticipated faults*. We assume faults exist, but do not know what or where they are. Hence the generality of the checks for any erroneous states that the faults may have caused. As an exaggerated example of a reasonableness check for unanticipated faults, consider a procedure that computes altitudes from pressure and radar sensor readings. A difference of several thousand feet between the new value and that derived on the previous call when the frequency of calls is on the order of milliseconds is wholly unreasonable. Even during a powered dive, aircraft simply do not change altitude that quickly. The check is not for any particular fault, nor does it pinpoint the fault. We have detected an erroneous state in the system and know that the cause may be somewhere in that procedure.

Damage Assessment

The act of damage assessment is the determination of how much of the system's state is affected. This may be through additional checking or assumptions about the communications structure or the system's structure. Assessing the damage to the state of a system (or component) can thus use information about the system's design. This method employs redundancy in that the implementation of the intended design is involved in what went wrong, but the intended design (a different "implementation" or statement of it) is involved in tolerating the faults of the first. Often, "damage confinement" is included under this activity. If affected information has been sent to other parallel activities, it may be possible to intercept it before those activities' states are also contaminated. If the damage can be contained or determined to be limited, less drastic recovery activities can be appropriate.

Error Recovery

The error recovery activity changes the error into a well-defined state that is not an error. Faults are not corrected. Faults are tolerated by correcting the errors they cause. Methods may be to restore a prior state, to impose another predesignated state, to transform the error into a valid state, or to create some state that might have been.

Forward error recovery is the construction of a new state from the error. This may be an ad hoc operation. Information about the specific check that detected the error and information gathered during damage assessment may be used. A forwardly adjusted state may itself turn out to be erroneous, necessitating either another (different) attempt at recovery or the signaling of another, larger ‘‘error’’.

Backward error recovery involves the re-use of a prior state, perhaps with some changes due to intervening changes to unrecoverable objects. A prior state must have been saved previously. We speak of both the code location and the point in time when a state is saved as a *recovery point*. A *recovery line* is a set of recovery points, one in each of several processes, that can be used to establish a prior state (or what could have been a prior state, had the scheduling been slightly different) for a system of processes.

Continued Service

Continued service allows the application to continue providing service. It also involves adjustment of control points to avoid the fault lest the error recur immediately. This is often achieved via reconfiguration. Using an alternate code section while avoiding the one blamed for the error may produce a lessened service, but part of the philosophy of software fault tolerance is that degraded or lessened service is preferable to no service at all. Reconfiguration may be replacement of a code segment. It may mean replacement of entire procedures with ones that use the same components under different algorithms or ones that use none of the same components. Whole processes or groups of processes may be destroyed and alternate processes started to take over the provision of their service. The replacement may be permanent or only for the current "use" of the affected subsystem. The idea behind non-permanent replacement is that the fault in the component may be limited to a few data combinations.

Design faults exist at execution because they were not discovered and removed before the system was placed into service. Thus, they are bound to be unanticipated. This fact drives the generality of many fault-tolerant strategies. Some of these stages are performed before the system is placed into service, both to save time and space and to reduce the complexity of determining a response at execution time. This a priori performance of fault tolerance also requires general-case responses to errors. For example, the N-version programming scheme detects errors when there is a minority in the vote on versions' results. Damage assessment rests on the assumptions that the minority is wrong and that no version was permitted to have non-local effects. Recovery is the same as if there were no errors. The results of the majority are used to construct a new state for use in subsequent activity.

As was mentioned, the initial thought of software tolerating faults in its own design seems ludicrous, but hardware fault tolerance has provided the model for software fault tolerance. For example, experience with hardware has given us replication, timing, reversal, coding, reasonableness, structural, and diagnostic checks [Ande81]. Since software faults are design faults, redundancy must be in terms of different designs. An *atomic action* (all or nothing computation) models the hardware fail-stop processor in that it either works as specified or it does not affect the remainder of the system. As we shall see, software fault-tolerance schemes proposed to date are modeled on hardware-fault tolerance strategies but are incomplete as guides for those who would build crucial systems.

1.5. Why Software Fault Tolerance?

Crucial systems as described above are being given serious consideration or are actually being built. Yet the people proposing and/or building them seem little concerned about or have little guidance in making the controlling software tolerant of its faults. We know humans *will* make mistakes in designing and implementing this controlling software. Proponents of software fault tolerance do not presume that we can forego debugging and testing of software. Every possible means is to be employed to eliminate the faults in the software. But the software's designers should also realize their fallibility, and the fallibility of fault-avoidance and fault-removal technologies. They should admit the inadequacies of the fault-avoidance effort, and prepare the software to tolerate the residual faults when they appear (as they are sure to do) once it is in operation.

Research into formal derivation of programs [Dijk76] and verification [VERk80, VERk81] and testing [Good75a] technologies is vital, yet there are real dangers in relying upon fault avoidance alone [Tane76]. If faults exist in a "proof" of a program, the program may still have faults. Naur developed a simple, one-page program with an informal proof [Naur69]. Later, some faults were found in the program, it was corrected and proved again [Lond71]. Still later, other faults were found, and a proof given of the revised version [Good75a]. Since then, still more faults have been found in the program [Myer78]. Another example of a faulty program with *two* invalid proofs is given by Geller [Gell78].

If the application of testing technologies is incomplete or test results are not thoroughly examined, faults may remain undetected until the software is placed into service. An experiment in testing [Myer78] found that, of three different testing technologies used by independent teams, none uncovered more than about a third of the faults. Some work has been done on developing a theory of testing [Good75a], but it is only a beginning.

The current design, testing, and verification technologies are not sufficiently advanced to rely totally upon them for crucial systems. Knight and Gregory [Knig83] have examined the available technologies and their shortcomings. Even were these technologies more advanced, their faulty application could allow faults to be created and to remain in a crucial system. A design for software tolerance of residual software faults could provide a last barrier to costly system failures.

1.6. Outline

The previous work in the area of software fault tolerance is reviewed in Chapter 2. Concepts that have been proposed previously are explained, and their disadvantages are examined. These are the problems that are well-known by the community and have been detailed in the literature. From that point on, we concentrate on backward error recovery.

Chapter 3 describes a syntax and the semantics of our solution to the problems identified in the previous work.

The programming language Ada® [Ada83] was designed specifically for embedded real-time applications and with the advice and cooperation of some of the best available talent in both language design and design of these applications. It is also expected to be the required vehicle of expression for a large subset of future applications in this area. Since Ada was specifically designed for these applications areas, it represents as good a choice as any real-time or concurrent programming language for illustrating ideas due to this and other software fault-tolerance research. Hence, Ada is used as the language of expression when illustrating concepts involved in this work. Further, Ada is expected to go through a revision in the 1990's. Thus, linguistic proposals due to this research are expressed as extensions or improvements to Ada. A certain familiarity with the semantics of Ada is assumed when examples of Ada text are given.

®Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

Chapter 4 describes an implementation of the solution presented in Chapter 3.

Once a solution to the problems of the previous work is achieved, we recognize other, more fundamental, problems. Chapter 5 describes these previously unidentified problems in general. They have to do with communication through shared data and process manipulation, and the structural constraints imposed upon processes by backward error recovery.

Chapter 6 examines the problems of communication through shared data and proposes a solution thereunto. Chapter 7 examines the problems of communication through process manipulation and proposes solutions for them. Chapter 8 examines the structural constraints imposed upon processes by backward error recovery. Some non-solutions to this problem are presented, and a potentially fruitful approach to a workable solution is suggested but not followed.

Chapter 9 summarizes what our strategies do for software fault tolerance and discusses where future extensions of this work might be expected to lead.

The appendices contain detailed information concerning the ideas presented in the chapters. They are not extraneous. This information is important, but too much detail was considered distracting from the main thrust of the new ideas. Appendix A details syntactic extensions to a successor to the Ada language that would include backward error recovery facilities based on Chapter 3 and Chapter 4. The syntactic description is accompanied by the specialized semantics of these facilities. Appendix B proves by example the contention that all of the previously proposed facilities for backward error recovery can be programmed as special cases of those described in Chapter 3, Chapter 4, and Appendix A. Finally, Appendix C provides an example application programmed using these facilities. This both demonstrates the use of the facilities of Chapter 3 and Chapter 4, and illustrates some of the further problems dealt with in the later chapters.

CHAPTER 2

Previous Work in Software Fault Tolerance

The previous work in the field of software fault tolerance is surveyed in this chapter. Most of the software fault-tolerance schemes discussed below contain very powerful ideas that could be useful in our own strategy although criticisms are raised. We do not condemn a software fault-tolerance strategy by listing its weak points; rather, this should be viewed as evidence that the problem of providing usable language facilities for implementing fault-tolerant crucial systems is unsolved. Extensive bibliographies on hardware and software fault tolerance are given by Anderson and Lee [Ande81], including the many applications areas excluded from consideration in this research.

2.1. Description of Previous Work

The major distinguishing characteristic among software fault-tolerance strategies is the mode of recovery. We describe forward error recovery schemes first, then backward error recovery schemes. There have been efforts that do not fit neatly into either of these categories. They may employ combinations of forward and backward error recovery, or their major contributions may have been concentrated upon other aspects of the problem. These are described separately.

2.1.1. Forward Error Recovery

Recall that forward error recovery involves creating a new system state based upon information in the erroneous system state and the analysis done in the damage assessment activity. *Exception* handling in an unorganized sense has actually been employed in many programming languages to modify an erroneous state into one that is hoped to be correct. The other major proposal in this category, N-*version* *programming*, involves creating a new system state based on the system's state augmented by a majority or consensus of the results of multiple versions of the software.

2.1.1.1. Exceptions

Exceptions [Good75b, MacL77] in programming languages are modeled on the *hardware interrupt*. A check in an underlying interpreter triggers a branch to a code section designed especially to deal with, or handle, that specific kind of situation. The handler is expected (to be able) to examine the whole machine's state to determine what needs to be done and to carry this out. The canonical example of an exception is an attempted division by zero. In exception handling, one attempts to deal with all possible faults by enumerating failure modes of components and providing one handler for each [Mell77].

Here we view an exception as a signal from a component of a system at some level. The signal indicates that a requested action was a violation of the interface between the using and used systems, and that the requested action has not been performed. That the action has not been performed does not imply atomicity, there is no guarantee that the used system was left in a consistent state.

Cristian advocates using exceptions for software fault tolerance [Cris82]. He deals in his own notation for exceptions. An example is given in Figure 2.1. Each procedure declares (Figure 2.1c) all exceptions that can be raised in the user of that procedure as a result of that use, and it is assumed that the programming language definition similarly lists all exceptions that can result from the use of any language construct. An exception can be raised explicitly (Figure 2.1d)[†]. A handler or set of handlers can be attached to each statement. Boolean (B) acceptability or consistency checks follow the form of Figure 2.1a, and other statements (O), which can be language constructs or procedure calls, follow the form of Figure 2.1b. If the exceptional condition (D) occurs during this use, control branches to the handler H, and follows the semicolon otherwise. Cristian says that the handler should undo what has

[†]Actually, instead of the keyword **RAISE**, he uses a symbol that is a tilted triangle with one corner pointing to the right. We find **RAISE** more instructive.

```
(a)        [B->H];

(b)        O[D->H];

(c)        PROC P SIGNALS E;

(d)        RAISE E;
```

Cristian's Exception Syntax

Figure 2.1

been done in the procedure so far if the procedure is going to signal an exception in its caller, i.e. the handlers in a procedure should make it atomic. Exit from H can be by having it simply end, allowing control to continue after the semicolon, or having it signal in the caller an exception declared by the surrounding procedure.

To accompany his exception notation, Cristian has created a *proof system* (Figure 2.2) for single-entry/multiple-exit programming language constructs and with data types other than "the integers" [Cris84] to help ensure that all possible cases are covered by the normal flow of the program and the flow through any exception handlers. In the proof system, a procedure's input space consists of three sets, called *domains*. The standard domain is the set of inputs for which the procedure can be shown to perform its intended service without signaling an exception. The expected exceptional domain is the set of inputs for which the procedure can be shown to signal an exception. The unexpected exceptional domain is the set of inputs for which no preparation has been made.

2.1.1.2. N-Version Programming

The idea of N-*version programming* [Aviz77, Chen78] is to apply software redundancy by actually allowing multiple versions of the software to execute in parallel and by voting on their results. The

SD ≡ Standard Domain
ED ≡ Anticipated Exceptional Domain
UD ≡ Unanticipated Exceptional Domain

Prove:
1) $\text{UD} = \varnothing$

2) $\forall\, i \in (\text{SD} \cup \text{ED})\ \text{EXECUTION}(i) = \text{INTENDED}(i)$

where EXECUTION is the service provided by the program
INTENDED is the intended or specified service

Cristian's Proofs

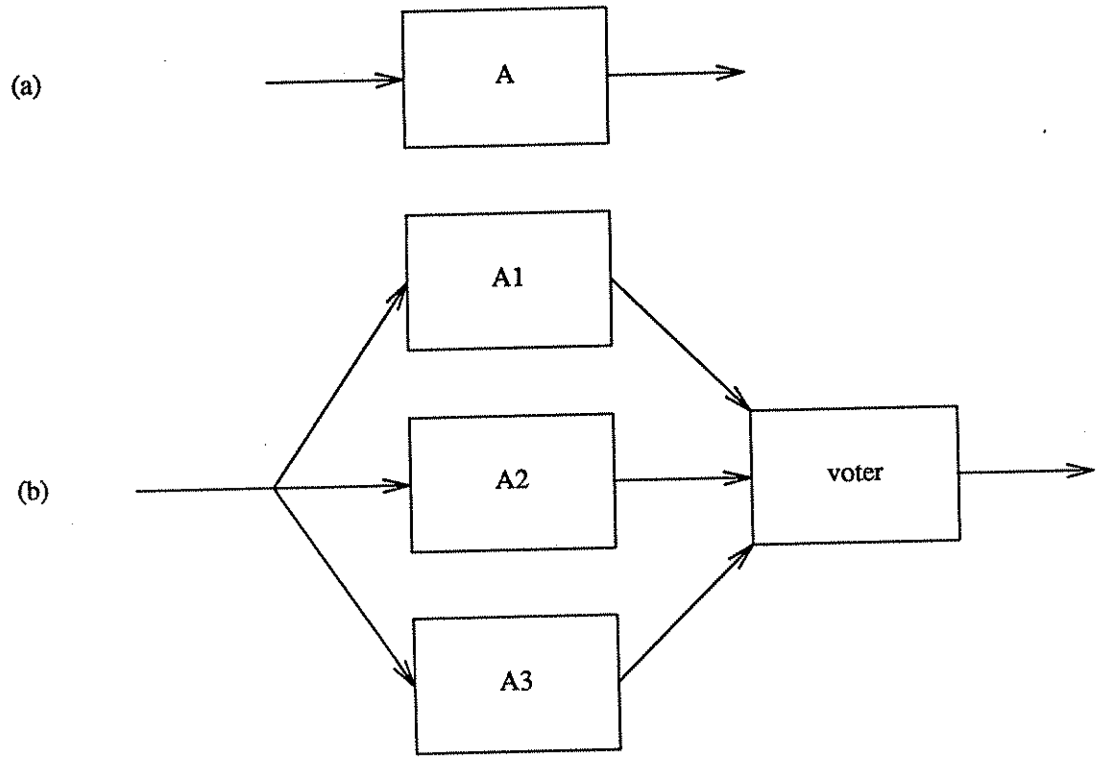
Figure 2.2

results of the majority are then used. N-version programming derives from the hardware fault-tolerance technique of N modular redundancy (NMR) (Figure 2.3). The multiple versions are all derived from the same requirements specification but, as far as possible, are designed and coded independently. The effort is to ensure that, although there will be faults in each version, no fault will exist in a majority of the versions.

The requirements specification is expected to give some indication of the kind of result that would be acceptable. An acceptability testing procedure is provided for each version to use before it allows its results to participate in a vote.

Each version executes in its own data space. The surrounding data space is only updated after the vote and only using the results computed by the majority of the versions.

The points at which a version submits results are called *cross-check* points. The results are formated into *comparison vectors* for use in the voting algorithm. Comparison vectors may contain *status flags* to indicate such things as a failure of the version's local acceptability test or that an exception was raised but not handled in this version, information that could save time for the voting mechanism by directing it to ignore results from this version.



A Component (a) and its Realization as 3-Versions (b)

Figure 2.3

An N-version programming system's operation is controlled by a *driver component*. A duty of the driver is to format inputs for the versions. If the versions are actually programmed in different languages, this might be extremely difficult. The driver must synchronize with each version to start it, to give it its inputs, to accept its results, and to destroy it should the version need to be replaced. Finally, the driver is responsible for performing a vote on the various results and updating the surrounding system state based upon the majority.

The most important part of an N-version program is the voting mechanism. A check for equality among results would only be appropriate for a small number of applications. Differing algorithms in the versions, and floating point calculations, contribute to a need for inexact voting techniques (see Figure

2.4). Besides arriving at a majority set of results, the voting mechanism must determine a single result, representative of the majority group, with which to update the system state. This may be as complicated as determining the majority set itself.

The versions whose results are in a minority may be removed from a system, or just have their results ignored for the current "use".

The N-version programming scheme detects errors when there is a minority in the vote on versions' results. This information is used for reconfiguring the set of versions, but has little effect on the immediate outputs. Recovery is the same as if there were no errors. The system's state and the results of the majority of versions of the software component are used to construct a new state for use in subsequent activity.

2.1.2. Backward Error Recovery

Recall that backward error recovery involves replacing the erroneous state by restoring a previous state that is presumed not to be erroneous or by creating such a state that might have existed previously. The major research efforts have produced *recovery blocks*, which organize this approach in sequential

```
(a) R := w1 × r1 + w2 × r2 + w3 × r3; (w1 + w2 + w3 = 1)

(b) R :=      if abs(r1-r2) < threshold then average(r1,r2)
         else if abs(r1-r3) < threshold then average(r1,r3)
         else if abs(r2-r3) < threshold then average(r2,r3)
         else signal error;
```

Adaptive (a) and Non-Adaptive (b) Voting for 3 Versions

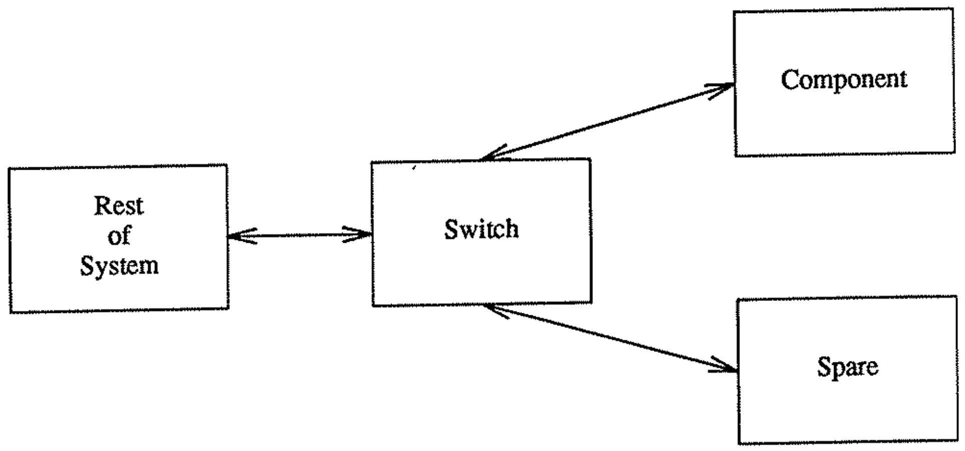
Figure 2.4

programs or program segments, and *conversations*, which attempt to make this approach amenable to use by parallel processes.

2.1.2.1. Recovery Blocks

A common technique for providing tolerance of hardware faults employs stand-by spares (Figure 2.5). This is the basic idea of a *recovery block* (Figure 2.6) [Horn74, Shri78a, Shri78b]. A recovery block encases redundant versions of a software component between a recovery point and a boolean acceptance test.

Upon entry to a recovery block, a recovery point is established by making a copy of the system's state. The *primary* version of the software component is then executed. At the end of the primary, the acceptance test is evaluated. If the acceptance test is successful (evaluates TRUE), the recovery point is discarded, thus *committing* the system to the effects of the primary version's execution, and control



A Component and a Stand-by Spare

Figure 2.5

```
<statement>      ::= ... | <recovery block>

<recovery block> ::=

     ENSURE <boolean expression> BY
        <primary>
   { ELSE BY
        <alternate> }
     ELSE ERROR
```

The Recovery Block Syntax

Figure 2.6

leaves the recovery block. Should the acceptance test fail (evaluate FALSE), the effects of the primary version's execution are discarded by restoring the system's state from the recovery point, and execution begins anew at the first *alternate* version of the component. At the end of the first alternate, the acceptance test is evaluated again. If the acceptance test is successful, the recovery point is discarded, and control leaves the recovery block. Should the acceptance test fail, the system's state is restored again from the recovery point, and execution begins anew at the next alternate. The process continues until the effects of one of the alternate versions pass the acceptance test or until the sequence of alternates is exhausted. If neither the primary nor any alternate version produces a state that passes the acceptance test, the system's state is restored from the recovery point and the recovery point is discarded; further, the recovery block statement signals failure of that component within the surrounding system.

When a recovery block's alternates are exhausted, none of the versions of the software component was able to perform the requested service. This could be due to an already erroneous state existing on entry to the recovery block, or due to improper use of the software component. In either case, the fault is external to the recovery block. Thus it is deemed appropriate to signal failure when a recovery block's alternates are exhausted.

Recovery blocks can be nested. If the nested recovery block signals its failure, execution of the primary or alternate in which it is nested ends. Subsequent actions are as if that primary or alternate had failed the acceptance test. Other component failures are similarly defined. For example, an unhandled exception raised within or propagated into the primary or an alternate is a failure of that primary or alternate. Should an exception occur during evaluation of the acceptance test, the exception is not handled or propagated. Rather, the primary or alternate for whose effects the acceptance test was being evaluated fails the acceptance test. If a control system's outermost recovery block signals failure, that situation must be treated as some form of anonymous exception.

The acceptance test is an overall check for acceptability of the state produced by execution of some version of the component as regards the goal of the statement. The fact that the alternates are allowed to provide degraded services with respect to the primary constrains the acceptance test. It must be possible for any results of the primary or any alternate (assuming they are correct) to pass the test, yet the test must be strict enough to detect errors produced by any of the primary or the alternates. The construction of acceptance tests is very application-specific, but a few attempts have been made to point the way for acceptance test construction [Lee78]. For example, an adjunct facility would permit retrieval of original values of variables. In Figure 2.7, whether or not the variable `speed` has been changed, the value of `PRIOR speed` would be the value of `speed` before the recovery point was established (the notation `x'` has sometimes been used for `PRIOR x`).

```
ENSURE
   abs(PRIOR speed - speed)/t <= (maxthrust + gravity)
  BY ...
```

Use of Prior Values in Acceptance Tests

Figure 2.7

Damage assessment is based on the assumption that no part of the state affected by execution of the current alternate is usable. The response is always to discard the error in favor of a previous state that is presumed to be correct.

The hardware *recovery cache* has been proposed to answer criticisms about the inefficiency of establishing recovery points and state restoration [Lee80].

Reconfiguration in a recovery block is achieved by replacing the faulty component with an alternate version of the component. The replacement occurs only for the current execution of the recovery block. On subsequent executions, the original primary is again attempted first. The reasons for this are that it would be difficult to specify statically the alternative dynamic reconfiguration, that the primary was written as the preferred algorithm for providing the component's service, and that the fault in the original version may be limited so that later executions with other input values would avoid it.

Alternate versions will have been constructed to perform the same or slightly degraded service as the primary. The service is "slightly degraded" if for no other reason than that some amount of time has been wasted in executing the primary. The alternates must not be copies of the primary. If they were, they would contain the same fault and in all probability create the same error. One proposed method [Lee78] of generating the alternates after the original primary is constructed, is to save all algorithms used during the development and maintenance of the system, making each new replacement the primary and demoting the old primary to first alternate.

Unrecoverable objects are variables whose values, unlike the rest of a system's state, cannot be restored by strict backward recovery. An example would be a variable at a memory address that has been mapped to a missile's firing mechanism. Once changed (say from zero to one), the missile has been launched, and no replacement of the original value in the variable will call it back.

Several approaches to dealing with unrecoverable objects have been proposed. For example, the actual unrecoverable objects could be hidden behind recoverable facades. In the missile example, the actual memory location would not be updated until execution had left the outermost surrounding recovery block. This would ensure that the control system was committed to the change before it took place.

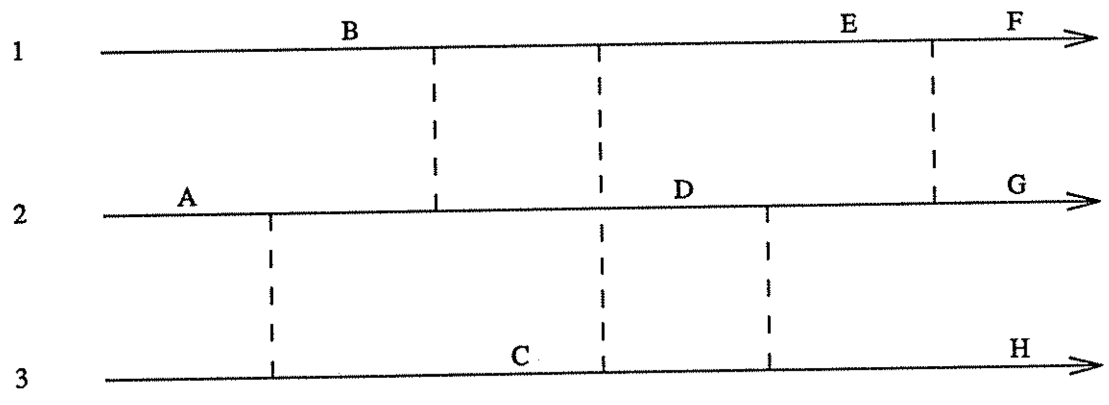
2.1.2.2. Conversations

The concept of a *conversation* [Rand75] is the canonical proposal for dealing with communicating processes via backward error recovery.

When two processes communicate, obviously information is passed between them. When two processes merely synchronize without explicitly passing data values, they still pass information. The information garnered 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 or 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 detected may not be in the process containing the fault. A solution to these problems is to roll back or perform backward error recovery on both processes. If the recovery points for all of the processes involved do not form a recovery line, the *domino effect* [Rand75] could result.

Figure 2.8 illustrates the domino effect with three processes 1, 2 and 3 progressing with time to the right, and recovery points A, B, C, D, and E. The vertical dashed lines represent communications between the processes. An error detected in 1 at F "causes" 1 to roll back to E and 2 to roll back to D. But this rollback invalidates information exchanged between 2 and 3, so 3 must also be rolled back to C.



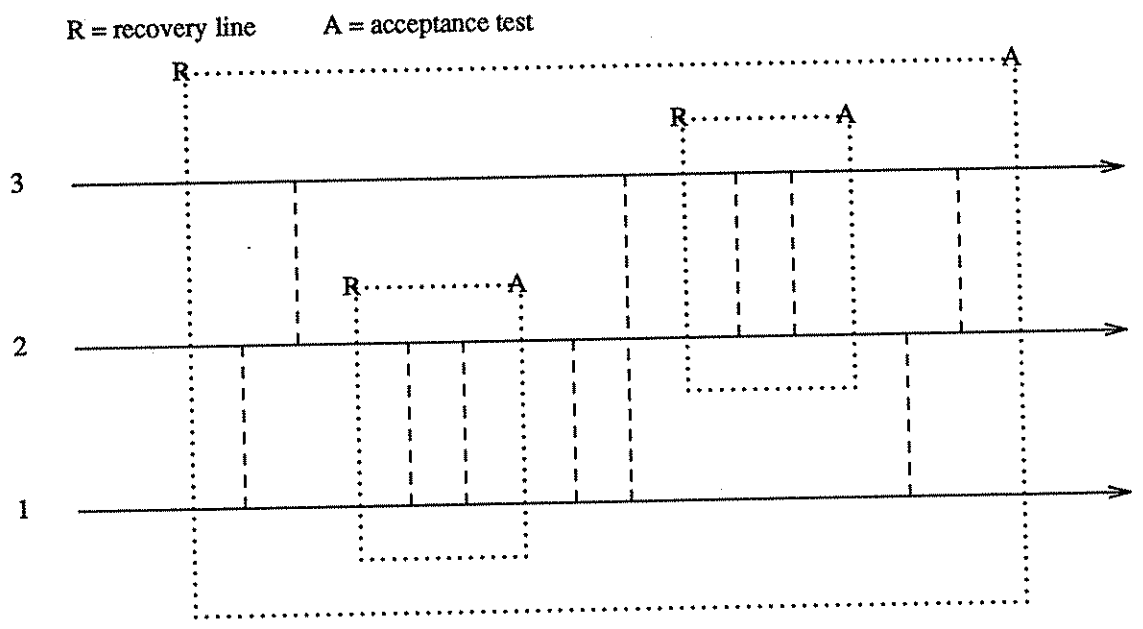
The Domino Effect

Figure 2.8

Since 1 and 3 communicated between points C and E, 1 must again be rolled back to B. The effect could conceivably spread to other processes and continue until the entire software system was rolled back to its initial state, thus discarding all information gathered during its operative life. Thus 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 use of conversations avoids the domino effect.

In a *conversation* (see Figure 2.9), a group of processes establishes a recovery line and they begin communicating. 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, 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.

Conversations may be nested as is illustrated in Figure 2.9. From the point of view of a surrounding conversation, a nested conversation is an atomic action. The encased activity seems either



Three Nested Conversations

Figure 2.9

not to have begun or to have completed, and no information that would be evidence to the contrary escapes. 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 1 while processes 2 and 3 were conversing, all effects since the larger recovery line (including the already-completed conversation between 1 and 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 allow enforcement of the rules. Russell and Tiedeman [Russ79] have proposed the *Name-Linked Recovery Block* as a syntax for conversations. As illustrated in Figure 2.10, the syntax appropriates that of the recovery block. What would otherwise be a recovery block, becomes part of the conversation designated by the conversation identifier. 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 conversant reaches the end of its primary or alternate. If any one or more of the processes fail their acceptance tests, all conversants are rolled back.

In other work [Russ77], Russell loosens the structure of conversations. He proposes 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 more rigid syntax. He gives three primitives for these operations: MARK, RESTORE, and PURGE respectively. They are all parameterized to designate the subject recovery point, and they apply to an individual process. This allows the programmer to save many states and restore the one he chooses, rather than the most recent. In contrast to the recovery cache, the recovery information saved by each MARK primitive must be complete. This is because recovery points are not constrained to be RESTOREd in the reverse of the order saved. The proposal assumes message buffers for inter-process communication. As part of backing a process up to a recovery point, previously received messages must be placed back into the message buffers.

```
CONV <conversation identifier> : <recovery block>
```

Russell's Conversation Syntax

Figure 2.10

Kim has examined several more possible syntaxes for conversations [Kim82]. His approaches assume the use of monitors [Brin77] as the method of communication among processes. He examines the situation from two philosophies toward grouping.

In one scheme, the *Conversation Monitor* (Figure 2.11), the conversing activities are grouped with their respective processes' source code, but are well marked at those locations. In another scheme, the *Conversation Data Type* (Figure 2.12), 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 he is addressing 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.

A third scheme, the *Concurrent Recovery Block* (Figure 2.13), attempts 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.

```
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 2.11

```
TYPE c = CONVERSATION( <conversation names > )
         PARTICIPANTS proca( <formal parameters> );
                      procb( <formal parameters> );
                      ...
         VAR cml : <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 cml,cm2...
         END

VAR convl : C;

                 (a)

convl.proca( <actual parameters> );

                 (b)
```

Kim's Conversation Data Type Syntax

Figure 2.12

2.1.3. Other Work

The *consensus recovery block* [Scot83] is an attempt to combine recovery blocks and N-version programming. A syntax would closely resemble that of a regular recovery block. The primary and all of the alternates are executed in parallel in their own data spaces and the results are voted upon. If there is a clear majority, the results represented by the majority are used. If there is no clear majority, the results of

```
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 2.13

the primary and alternates are used in evaluating the acceptance test with preference following the listed order. The first set of results that passes the acceptance test is used. The scheme affords a second chance after an inconclusive vote and avoids reliance upon a potentially faulty acceptance test unless absolutely necessary.

In an attempt to adapt conversations to the structure of many actual real-time applications, Anderson and Knight proposed *exchanges* [Ande83b]. An exchange is a conversation in which all of the communicants are created at the recovery line and destroyed at the acceptance test. The real-time program structure being applied is the *cyclic executive* under which time is divided into "frames". Inputs are accepted at the beginning of each frame. Outputs are produced at the end of each frame. 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 used for the current frame, e.g. those of the previous frame. The only information saved at the "recovery line" is that needed to provide the alternate outputs.

Campbell, et al proposed what is called the *Deadline Mechanism* [Camp79]. 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), the alternate algorithm will be. Several simulation studies have been performed showing a reduction in timing failures when such a mechanism is employed [Camp79, Wei80, Lies83].

A field related to our problem is *software safety* [Leve82]. The work being done on *software fault tree analysis* [Leve83] could be of use in developing acceptance tests. As many of the potential physical hazards are identified as is possible, then a fault tree analysis is performed. "The basic procedure is to assume that the software has performed in a manner that [...] will lead to a catastrophe and then to work backward to determine the set of possible causes for the condition to occur." ([Leve83] p.571) Fault tree analysis could be used to determine where software checks need to occur to avoid safety hazards, and the study of software safety can influence what forms of degraded service are appropriate for a particular application.

2.2. Shortcomings of Previous Work

A general consideration for crucial systems is time. Acceptance tests and voting codes must be reached and reached on time for the results to be useful at all. The *unexpected delay problem* is that some unanticipated circumstance may cause a particular section of code to be executed too late for its results to be useful or not to be executed at all. For example, an infinite loop would cause a section of code not to be executed. A real-time program need only be late to be considered faulty. A scheme for providing software fault tolerance must address the unexpected delay problem.

Another consideration is the management of *complexity*. If the use of a scheme involves too much effort on the designer's (programmer's) part, it may be counter-productive in that more faults will be generated through the use of the scheme than would otherwise occur. Furthermore, a fault in the application of the scheme might make the system more dangerous than if fault-tolerance efforts had not been applied at all. A scheme supported by a rigid, encasing, structured syntax allows design-time (compile-time) enforcement of the accompanying semantic rules. Such a quality in a scheme allows for added complexity without added faults.

2.2.1. Forward Error Recovery

A major drawback of any forward error recovery scheme so far proposed is the complexity that must be involved in creating a new state for the system from the ruins of an error. Because a state is so specific to the application system, this activity must be done under the application's direction. There is very little a language support system can do to help. There are few rules to enforce, and it is difficult to determine what special functions should be made available whose misuse would not bring about immediate failure of the entire control system. The code that directs the new state generating activity is as subject to faults as the code whose faults caused the error. Further, the information with which this code must work is, by definition, erroneous.

Neither exceptions nor N-version programming provides a solution to the unexpected delay problem. Nor is either a concept or construct for dealing with parallel programs. Proponents of backward error recovery view exceptions as complimentary to the backward error recovery approach rather than sufficient for tolerating faults [Mell77]. There have been experiments which call into question a basic assumption of N-version programming — the independence of versions [Neum81, Knig85].

We will not pursue forward error recovery further. Our focus will be upon backward error recovery.

2.2.2. Backward Error Recovery

Backward error recovery seems [Mell77] to be the more general approach (than forward) since the error is thrown away in favor of what is presumed to be a valid state. On the other hand, if the latency interval between the first error due to a fault and the first detection of an error is beyond a commitment interval, none of the backward recovery schemes so far proposed can handle the error [Cris82].

2.2.2.1. Recovery Blocks

In a recovery block, there is only one test for acceptability of results. How to program the acceptance test to be both meaningful and allow a wide range of alternate algorithms to pass it is unspecified. Design diversity in the primary and the alternates, combined with the possibility of degraded service from the alternates, implies that the acceptance test must not be made very strict. It must be possible for any results of the primary or any alternate (assuming they are correct) to pass the test, yet it must be strict enough to detect errors produced by any of the primary or the alternates. This combination may not be possible. A test that is general enough to pass all valid results might not be specific enough to actually detect all errors within the construct. 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. The recovery block really needs multiple tests, one for the primary and one specific to each of the alternate algorithms†, perhaps with a

†This has been noted previously and an inelegant ad hoc solution proposed [Lee78].

general overall test as a check on the various individual tests.

The recovery block is strictly a sequential programming construct. It gives no hint about recovery after inter-process communication. The conversation concept was an appropriation of the recovery block concept, not an integral part.

There is the question of when a recovery block should be used. There is little indication as to what portions of a program should be protected by recovery blocks. If used on every routine and every statement sequence, the tests may become trivial and fail to offer any benefit. If recovery blocks are only used at the outermost levels, the acceptance tests may be so complex as to duplicate the complexity of the primary or alternates. This may introduce more faults in the acceptance test than the primary alone, or it may squander processing resources so that execution of an alternate would bring about a timing failure.

The unexpected delay problem, specifically the timing of control program activities, has remained largely unaddressed by the recovery block scheme. It has only been mentioned once [Hech76], and not incorporated into subsequent writings on the subject.

We must rectify the use of unrecoverable objects with the backward recovery strategy. As was pointed out earlier, there is some discussion in the literature on how recovery blocks could be reconciled with nested recovery block commitment to unrecoverable objects. A useful scheme will not ignore this issue.

The problem of the latency intervals for fault detection being longer than commitment intervals is not addressed. That is related to the problem of how to construct meaningful acceptance tests. It is assumed that acceptance tests can be constructed that can detect errors before they become so widespread, or that multiple layers of nested recovery blocks' acceptance tests can together detect them. The possibility of nested recovery blocks allowing such errors to ''escape'' should not be permitted.

How acceptance tests should be designed is not made clear in the literature, although a few attempts have been made to point the way [Lee78]. We do not advance that work beyond suggestions about incorporating the software safety work. The problem of acceptance tests being subject to single-point failures could be alleviated by the nesting of recovery blocks and the inclusion of separate tests for each algorithm within a recovery block.

2.2.2.2. Conversations

Major shortcomings of the conversation scheme and of all its follow-up syntactic proposals lie in the acceptance test(s). Here we find the same shortcomings we found in recovery blocks. 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. An example of diverse strategies will appear shortly.

Another shortcoming 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, as in the original conversations proposal, 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 acknowledge 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 if they have deadlines to meet, as is likely in a crucial system. Thus, desertion is another form of the unexpected delay problem. The processes in a conversation must be extricated 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 even addresses the desertion problem. The solution there is to enclose the entirety of each participating process within the conversation. Not only cannot a process fail to arrive at a conversation, it cannot exist outside of the conversation.

The original conversation proposal made no mention of what was to be done if the processes ran out of alternates. Two presumptions may be made: that the retries proceed indefinitely, which is inappropriate for a real-time system, or that an error is to be automatically detected in each of the processes, as is assumed in all of the proposed conversation 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 presentations of 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, that may not necessarily be true if processes are allowed to converse with alternate groups.

Russell's work permitting the application to have direct control over establishment, restoration, and discard of recovery points has its own set of problems. First of all, his premise ignores the possibility that the information within a message can contaminate a process' state. When the receiver of a message is rolled back, he merely replaces the same message on the message queue. Russell's application area is that of producer-consumer systems. The control systems we are considering are often 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. Russell's scheme allows completely unstructured application of the MARK, RESTORE, and PURGE primitives. This fact, along with the complicated semantics of conversations, which they are provided to create, affords the designer much more opportunity to introduce faults into the software system. For example, it is noted that 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.

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. Since the name-linked recovery block proposal makes no mention of the method of communication among processes within a conversation, it remains open to charges of permitting smuggling. 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 this and all subsequent alternates.

The concurrent recovery block is not even 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.

Most other shortcomings are more like open questions for the designer. When should a conversation be used? How should one determine which processes should be included in a conversation and which should use their own separate conversation? There is little discussion in the literature on how conversations could be reconciled with nested conversation commitment to unrecoverable objects. Nor is the problem of the latency intervals for fault detection being longer than commitment intervals addressed.

Related to that is the recovery block problem of how to construct meaningful acceptance tests. Smuggling in conversations and the unexpected delay problem also remain unresolved.

2.2.3. Other Work

The consensus recovery block requires more resources than either N-version programming or regular recovery blocks. Considered as an N-version programming variant, it imposes the additional cost of deriving the acceptance test. Seen as a recovery block variant, it requires the alternates to be designed, not for degraded service, but for the same level of service. This is because their results will be voted upon to find a majority. Further, a voting algorithm must be designed. Also, more processing time will be expended on average since the primary and all alternates must be executed to completion before the results of any of them are checked for usefulness in the surrounding environment. And, like both the recovery block and N-version programming schemes, this scheme still fails to address the unexpected delay problem.

The idea of exchanges has direct utility only in systems employing the scheduling regime known as the cyclic executive. 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 [Dijk68], which may not always be suitable. For example, it becomes very difficult to program multiple frame rate systems, the first variation that is often imposed on the cyclic executive theme [MacL80].

Campbell's 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. Does each get a complete copy as in N-version programming, or is there a form of backward recovery? Either method would require some time that must be considered in the satisfaction of the timing request. This proposal focuses too narrowly upon only one issue, that of timing, and provides incomplete coverage of that.

To mention software safety again, we do not consider it a panacea for the problem of constructing acceptance tests, nor for that of determining placement of fault-tolerance constructs within an application. The safety hazard analysis is every bit as likely to be prone to human error as is any other design method.

2.3. Summary

The overall problem is to create a strategy for organizing the design of the software for real-time concurrent systems so as to enable it to tolerate its own faults. A realistic, comprehensive approach to software fault tolerance has never been brought forth that could be used in guiding designers of the applications that are most in need of it. As we have seen, systematically thought-out software fault-tolerance proposals are all lacking and/or out of touch with how real systems are constructed or with their requirements. The first step is to adapt to reality the systematic proposals that do exist while answering objections to them and correcting for their problems. Shortly, we show how to adapt these as facilities in a programming language to aid the designer (programmer) in applying backward error recovery. Only then will it be possible to try building systems with software fault tolerance in mind. This will in turn permit the determination of appropriate placement of these facilities and evaluation of the efficacy of the backward error recovery approach to fault-tolerant software.

The shortcomings of the previous work generalize into eight issues.

(1) A general consideration for crucial systems is time. Boolean acceptance tests and voting codes must be reached and reached on time for the results to be useful at all. If a scheme does not

address the unexpected delay problem, then it is insufficient for providing software fault tolerance in a real-time program since a program in that context needs only to be late to be considered faulty. Time also needs to be considered in designing the programming language facilities to support a software fault-tolerance strategy.

(2) Another consideration for a fault-tolerance scheme is the management of complexity. If the use of a scheme involves too much effort on the designer's (programmer's) part, it may be counter-productive in that more faults will be generated through the use of the scheme than would otherwise occur. Furthermore, a fault in the application of a fault-tolerance scheme might make the system more dangerous than if fault-tolerance efforts had not been applied at all.

> [A] well-structured approach to the design and inclusion of fault-tolerance techniques is a prerequisite for their success; an unstructured approach could easily reduce system reliability by introducing more faults than those to which tolerance was provided. ([Ande81] p.15)

A scheme supported by a rigid, encasing, structured syntax allows design-time (compile-time) enforcement of the accompanying semantic rules. Such a quality in a scheme allows for added complexity without added faults. By embedding the strategy within the programming language, we can shield the programmer from much of the added complexity.

(3) A scheme must deal with parallel processes. A control system based on a sequential program could be a very poor design and would certainly be limited. The scheme should not be overly restrictive of the autonomy of processes, as is the forced retry of the conventional conversation.

(4) A complete scheme will have its semantics fully specified. Many of the shortcomings we noted for the individual schemes in Chapter 2 have to do with incomplete specification. An incompletely specified scheme allows mistakes to be made in its use, rendering it ineffective if not dangerous.

(5) The overall strategy may involve specialized features to be placed at different levels within a system. A scheme should be accompanied with instructions for placement of its various parts

within the control system.

(6) A scheme must take the constraints of real applications into account, even take advantage of them if possible. One of the constraints we have already identified is the existence of real-time deadlines.

(7) A scheme must address the problem of dealing with unrecoverable objects in a fault-tolerant manner. The control system must deal with sensors and actuators as unrecoverable objects. This is a very special case of taking reality into account, but deserves separate attention. The goal of fault tolerance is to prevent failures of the control system. Erroneous settings of unrecoverable objects can be nothing other than failure.

(8) A scheme must deal with the problem of the latency interval between the first error due to a fault and the first detection of an error being beyond a commitment interval.

Of these, issues 5 and 8 are not addressed within this work. In particular, solution of issue 5 must await experimentation with the solutions to the other issues as proposed in the following chapters. Issue 7 is addressed somewhat in Chapter 6. The aspects of issue 1 relating to efficiency of the scheme itself must await refinement in future work as must the aspects of issue 2 relating to programming of acceptance tests.

CHAPTER 3

A Unifying Structure for Fault-Tolerant Software

In this chapter, we introduce a new building block for concurrent programs called the *dialog* and a new backward-error-recovery primitive called the *colloquy*. The colloquy is constructed from dialogs. Together, they remedy the various limitations of the previous fault-tolerance proposals. The previous proposals are special cases of the colloquy. Thus, the colloquy provides a general framework for describing backward error recovery in actual crucial programs.

The dialog and colloquy are proposed as general concepts but a specific syntax for their use is given as extensions to Ada. That syntax borrows from that language and is detailed in Appendix A. This syntax has been chosen for illustrative purposes. The actual syntax is irrelevant; the concepts could be used in many other programming languages. However, once chosen, a rigid syntax can allow a compiler to enforce certain of the semantic rules. Such enforcement by a language processor has the potential to catch additional faults which might otherwise be introduced through the use of a complex framework.

3.1. Overview

There are several concepts not previously employed effectively in backward error recovery strategies. These are identified in the following paragraphs before their specific application to the dialog and colloquy is discussed.

Recall that desertion is the failure of a process to enter a communication when other processes expect its presence. The process might enter the communication and still fail to arrive at its exit, thus preventing others from exiting. Whether the process will never enter the communication, is simply late, or enters the communication only to take too long or never arrive at the acceptance test, does not matter to the others if they have real-time deadlines to meet. Each process may have its own view of how long it is willing to wait, especially since processes may enter a communication asynchronously. Whether they protect inter-process communications or sequential parts of processes, acceptance tests must be reached and reached on time for the results to be useful. Meeting real-time deadlines is as important to providing the specified service as is producing correct output. In order to deal effectively with desertion, especially in critical systems, some form of timing specification on communication and on sequential codes is vital.

After an error during a communication activity is detected, a process should not be forced to continue to re-try communication with the same set of processes. We will use conversations to illustrate this. When it needs to communicate, a process enters a conversation and stays there, perhaps through many alternate algorithms, until the communication is completed successfully. The same group of processes are required to be in the alternate interactions as were in the primary. The recovery action merely sets up the communication situation again. In the original form of conversation, once a process enters the construct, it cannot break out but *must* continue trying with the same set of other processes, including one or more which may be incapable of correct operation. The rules for conversations force the saved state in each process to be one present after the process has committed to the beginning of communication. Hence, upon rollback after failure of an attempt, a new attempt with the same set of participants begins. The only difference is that the process may use a different algorithm in the new attempt. In practice, 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; in fact, different processes might make different numbers of attempts at communicating. Conversations do not allow this, although it would not be desertion were it systematic and intended. By reordering the entrance into a beginning-of-communication state and the establishment of recovery points, we can introduce the possibility of processes communicating with entirely different groups of other processes during alternate attempts than during the primary.

In a conversation or recovery block, once rolled back and reconfigured, a process attempts to achieve its goal again, and eventually reaches the *same* acceptance test again. True *independence* of algorithms between primary and alternates, within the context of backward error recovery, might require very *different* acceptance tests for each algorithm, particularly if some of them provide significantly degraded services. A single test for achievement of a process' goal at a particular point in its text would of necessity have to be general enough to pass results of the most degraded algorithm. This might be too general to enable it to catch errors produced by other, more strict, algorithms. These considerations suggest the need for separate acceptance tests specifically tailored for each of the primary and alternate algorithms.

It must be remembered that, although each process has its own reasons for participating in a communication activity, there is a goal for the *group* of processes as well. Rather than combine the individual goals of the many participants with the group goal in a single acceptance test (perhaps allowing the programmer to forget some), and rather than replicating the test for achievement of the group goal within every participant, there should be a separate acceptance test for each participant and another for the group.

A final problem with many previous backward error recovery proposals is that if a process runs out of alternates, no scheme is provided or mentioned for dealing with the situation. Such a scheme has not only to be provided but to be enforced so the programmer must explicitly (consciously) avoid it for those cases in which that is legitimate. In fact, exercise of any option should be made explicit.

3.2. The Dialog

A *dialog* is an occurrence in which a set of processes:

(a) establish individual recovery points,

(b) communicate among themselves and with no others,

(c) determine whether all should discard their recovery points and proceed or restore their states from their recovery points and proceed, and

(d) follow this determination.

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. Notice that *nothing* is said about what actions the processes take after the dialog. Specifically, nothing is said about actions to be taken in the case of failure.

It follows from the definition that 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 3.1 shows a set of three processes communicating within a dialog.

If we want a program to use dialogs, we need a syntactic facility for enforcing that protocol. To meet the definition, the syntactic facility must address the following issues:

(1) membership in the set of dialog participants,

(2) establishment of individual recovery points,

(3) communication among participants,

(4) communication with no other processes,

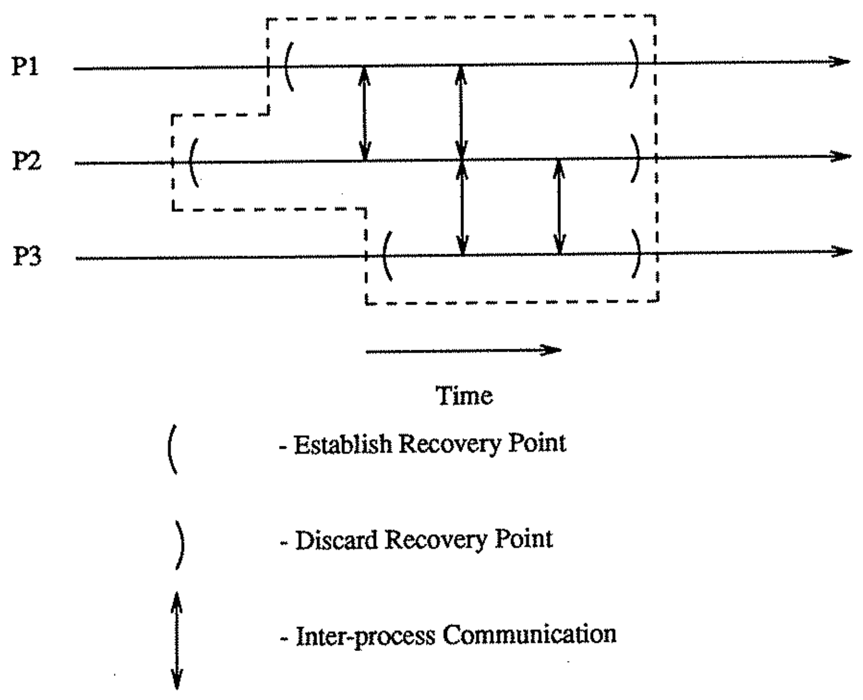
(5) how to determine whether to succeed or fail,

(6) restoration or discard of recovery points,

(7) ensuring proper nesting,

(8) what to do after making the determination.

We introduce the *discuss* statement as such a syntactic facility. The activities of a given process relative to a dialog would be denoted by a DISCUSS statement. Figure 3.2 shows the general form of a DISCUSS statement.



P1
P2
P3
Time
(- Establish Recovery Point
) - Discard Recovery Point
- Inter-process Communication

Three Processes Communicating in a Dialog

Figure 3.1

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

A DISCUSS Statement

Figure 3.2

A recovery point is established for a process as it begins execution of a DISCUSS statement (issue 2), and state restoration or discard of the recovery point occurs as the process "leaves" the DISCUSS statement (issue 6). We assume that the machine(s) on which programs are to execute can be made to establish, restore from, and discard recovery points on a process-by-process basis. This may be by a recovery cache or one simulated by compiler-generated software. The mechanics of this for a single process are discussed elsewhere [Lome77, Lee78].

The syntax itself ensures proper nesting (issue 7) of dialog entry and exit for any one process. Chapter 4 describes a method by which an implementation can ensure that proper nesting of dialogs is maintained across processes.

The dialog name associates a particular DISCUSS statement with the DISCUSS statements of the other processes participating in this dialog (issue 1), *dynamically* determining the constituents of the dialog. This association cannot in general be known statically. 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.

We stated that the participants in a particular dialog cannot be known statically. There may be, say, three processes whose texts contain references to a particular dialog name. If two of them enter a dialog using that name, questions might arise about participation of the third. The third process may be executing some other portion of its code so that it is unlikely to enter a dialog of that name in the near future. If the two processes reach and pass their acceptance tests, they, being the only participants in the dialog, can leave it — the third process is not necessary to the dialog, so is not a deserter. If the dialog fails due to an acceptance test or a timeout (see below), the problem is not guaranteed to be the absence of the third process, so again it is not necessarily a deserter. If the dialog has no time limit specified (see below), that had to be explicitly stated by the programmer, so the two processes becoming "hung" in the dialog while waiting for the third was *not* unplanned.

A `dialog_name` may be re-used. If two groups of processes use the same `dialog_name` at the same time, they are in the same dialog. If one group's dialog has finished (successfully or otherwise) before the other group's members enter `DISCUSS` statements using that dialog name, the other group are in a different dialog and are said to be in another *instance* of the dialog.

The `sequence_of_statements` in the `DISCUSS` statement represents the actions which are this process' part of the group's actions within their dialog. Any inter-process communication, which must be via the normal language facilities (issue 3), *must take place within* this sequence of statements (i.e. must be protected by a dialog). The `DISCUSS` statement fails (issue 5) if an exception is raised within it, if an enclosed `dialog_sequence` (see below) fails, or if any timing constraint is violated.

The `Boolean_expression` is an acceptance test on the results of executing the sequence of statements (issue 5). It represents the process' *local* goal for the interactions in the dialog. It is evaluated after execution of the sequence of statements. 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.

The dialog names used in `DISCUSS` statements are required to be 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. It must be the name of a parameterless boolean function defining the global acceptance test. Being parameterless and boolean, there can be no overloading (as in Ada) to be resolved. This function is not allowed to have side-effects. The names mentioned in the `name_list` are the names of *shared* variables which will be used within dialogs that use this dialog name. This includes variables used within the function that implements the global acceptance test. Of non-locals, only a variable so named may be used within a DISCUSS statement, and then only within DISCUSS statements using a dialog name with that variable's name in its dialog declaration. The significance of these rules is that the set of shared variables can be locked by the compiler and execution-time support system to prevent smuggling (issue 4). In effect, the actions of the dialog's participants are made to appear atomic to other processes with respect to these variables. (Our implementation, described in Chapter 4, also prevents smuggling via messages or rendezvous).

The Boolean function named by the dialog name is evaluated after all processes in the dialog have evaluated their respective Boolean expressions and they all evaluate to TRUE. It is only evaluated once for an instance of the dialog; i.e. it is not evaluated by each participating process. Thus no process can leave a dialog until all processes currently in that dialog leave with the same success, and success involves the execution of both a local and a global acceptance test.

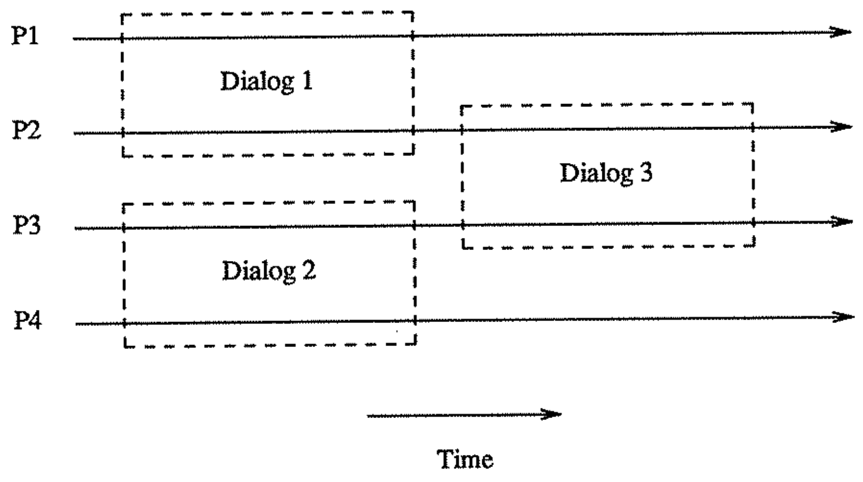
Issue 8, above, remains to be addressed by a protocol organizing the use of dialogs as expressed in DISCUSS statements.

3.3. The Colloquy

A *colloquy* is a semantic construct that organizes the use of dialogs to solve the problems of conversations. A colloquy is a collection of dialogs as defined below. At execution time, a dialog is an interaction among processes. Recall that 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 after a failed dialog, each process may make another attempt at achieving its original local goal, or some *modified* local goal through entry into a *different* dialog.

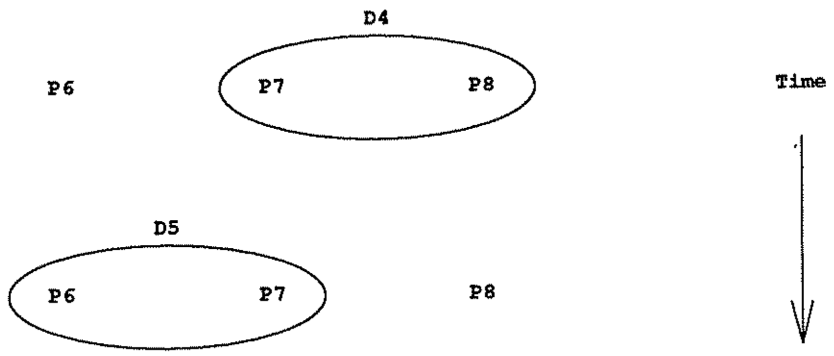
Each of the former participants of the now defunct dialog may choose to interact with an *entirely separate* group of processes for its alternate algorithm. The altered constituency of the new dialog(s) most certainly requires new statement(s) of the original global goal. The set of dialogs which take 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 3.3. A *discussion* is the subset of dialogs within a colloquy which involve any one process. In Figure 3.3, dialogs 1 and 3 comprise a discussion for process P2, while dialogs 2 and 3 comprise one for process P3.

That a colloquy and a discussion are not the same thing is illustrated in Figure 3.4. Processes P7 and P8 enter dialog D4 which fails. For its alternate algorithm in attempting to achieve its goal, process P7 enters dialog D5 with process P6. Both dialogs D4 and D5 are part of the same colloquy. Indeed, if process P8, in an alternate algorithm to that employed in dialog D4, enters a dialog with some other process, that dialog would be included as well. Similarly, if dialog D5 was entered by process P6 as part of an alternate algorithm to that employing a dialog with some other process, that dialog would be



Four Processes in a Colloquy of Three Dialogs

Figure 3.3

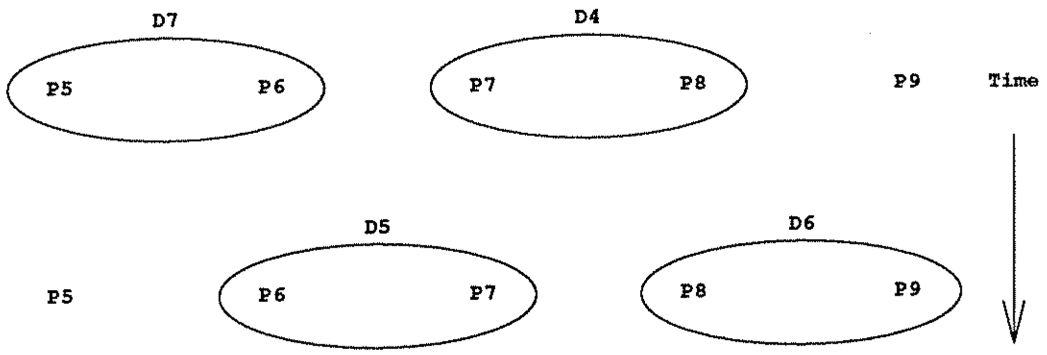


A Colloquy

Figure 3.4

included also. These latter possibilities are shown in Figure 3.5.

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



Another Colloquy

Figure 3.5

The general form of a `dialog_sequence` is shown in Figure 3.6. 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 3.6 by `attempt_1`. The attempt is actually a DISCUSS statement followed by a sequence of statements. To ensure proper nesting of dialogs and colloquys, a DISCUSS statement may appear only in this context. If the performance of these activities is *successful*, control continues with the statements following the `dialog_sequence`. The term "success" here means that no defensive, acceptability, or timing checks occurring within the attempt detected an error, and that no exceptions (if the language has exceptions) were propagated out to the attempt's DISCUSS statement. 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 locus of its text. These are exactly the algorithms which the process employs as its part of its discussion within the colloquy.

```
SELECT
      attempt_1
   OR
      attempt_2
   OR
      attempt_3

   TIMEOUT simple_expression
      sequence _of_statements

   ELSE
      sequence _of_statements
END SELECT;
```

Form of a `dialog_sequence`

Figure 3.6

It is a natural optimization for a structure controlling a (potential) sequence of dialogs to factor out the saving and discard of recovery points to the beginning and end of the sequence, leaving only state restoration between dialogs. We will continue to describe the `dialog_sequence` as if that optimization has been applied.

Exhaustion of all attempts 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`. Absence of a timing constraint must be made explicit by replacing the simple expression with the keyword NEVER. 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. Should an attempt achieve success or the list of attempts be exhausted without success before expiration of the interval, further actions are the same as for `dialog_sequence`s without timing specifications.

However, 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_sequence`s. 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.

[†] Success, here has to do with absence of raised exceptions or failures of nested `dialog_sequences`, since there is no acceptance test associated with the last-ditch algorithm.

As with the ELSE part, the TIMEOUT part allows the programming of a "last ditch" algorithm for the process to achieve its goal. The last-ditch algorithms are really a form of forward recovery since their effects will not be undone, at least at this level. If the sequence of statements in the TIMEOUT part is successful[†], control continues after the dialog_sequence. If not, or if there were no statement sequence, the surrounding attempt fails.

If any participant in a dialog has a timing constraint, the sequence of statements of the DISCUSS statement must be executed successfully and the boolean expression of the DISCUSS statement must be evaluated TRUE in each participant, and the common goal must be evaluated TRUE before the interval expires for the then current attempts of the participants to be successful. If several participants have timing constraints, the first interval to expire is the one to be "beaten" for success to be achieved. Expiration of one process' timing constraint interval while that process is executing a DISCUSS statement causes all processes currently executing DISCUSS statements with the same dialog name as this process to react as if the boolean expression in one of their DISCUSS statements had been evaluated FALSE, except that this process continues execution with the sequence of statements in the TIMEOUT part.

In any attempt, a statement sequence which is logically outside the dialog_sequence can follow the DISCUSS statement to provide specialized post-processing after the recovery point is discarded if the dialog succeeds. It is not subject to this dialog_sequence's timing constraint.

The programmer is reminded by its position after the TIMEOUT part that the ELSE part is not protected by the timer, and that it is reached only after other (potentially time-consuming) activities have taken place. The structure of the dialog_sequence also requires no acceptance check on these activities. The implication of these two observations is that the last ditch activities need to be programmed very carefully.

[†] Again, success here has to do with absence of raised exceptions or failures of nested dialog_sequences, since there is no acceptance test associated with the last-ditch algorithm.

A *fail* statement may occur only within a sequence of statements contained within a dialog_sequence. Execution of a FAIL statement causes the encompassing dialog to fail. The FAIL statement is intended for checking within an attempt. For example, it can be used to program explicit defensive checks on inputs such as:

```
IF input_variable < lower_bound THEN
   FAIL;
END IF;
```

It can also be used to simplify the logical paths out of an attempt should some internal case analysis reach an "impossible" path. With the FAIL statement, the programmer does not have to make the code for the attempt complicated by providing jumps or other paths to the acceptance test or to insure that some part of the test is always FALSE for such a special path. The FAIL statement can also be used to provide sequences of statements for the ELSE and TIMEOUT parts that make failure explicit rather than implicit (recall, implicit failure is indicated by their *absence*).

3.4. Discussion of Details

The colloquy is a very flexible and powerful concept. Fitting the concept into a real language presented many options only some of which we have chosen for the presentation above. That presentation focused on how the colloquy addresses the problems with the previous proposals. Other options and decisions which we considered interesting or important to provision of a fault-tolerance facility are discussed below.

3.4.1. Dynamic Association

Use of dialog names is a dynamic association, not a static one. This is analogous to entry calls and accept statements. In general, there is no way to tell statically which entry call in which process will be accepted by any particular accept within another process. Some processes may enter and leave instances of dialogs cyclicly, while others may join for a few or one instance only. There can even be varying numbers of processes in each instance of the dialog. This flexibility allows the programmer to design a process with differing sets of primary and alternate algorithms to be used as execution-time conditions may dictate. For instance, the process may be able to recover from an internal error by degrading itself based upon some internal test and by subsequently checking a corresponding variable. This kind of recovery involves no changes to the other communicants. If recovery from some error involves the services of several processes (one process) being taken over by one (several), the replacement(s) might subsequently be able to communicate with the remainder of the system without change to the remainder of the system.

3.4.2. Optional Statement Sequences in TIMEOUT and ELSE Parts

An absent "last ditch" algorithm for timing specifications is defined not to mask failure of the current discussion for the process which timed out. This is for the same reasons as will be described for the ELSE part below, to wit: to catch errors of omission.

The ELSE part of a `dialog_sequence` provides a place for a last-ditch effort at achieving the process' goal for the discussion. It would seem that an ELSE part with a null statement sequence would suffice for the case in which nothing can be done. But when nothing can be done, it is desirable to have a means of "propagating" a discussion's lack of success to a larger, surrounding context. This failure of the discussion is provided by the absence of the sequence of statements in the ELSE part. It is even more clear to require the ELSE part and have the programmer include an explicit FAIL statement within it. To rely on that would leave programs open to faults of omission in that the programmer might include some statements but forget the FAIL statement itself. Faults involving omitted program parts are more easily caught by defining execution of an absent optional part as failure than by requiring an explicit annunciation of that condition when the announcement (FAIL statement) might be accidentally omitted.

3.4.3. Un-Timed Statement Sequences

The sequences of statements in the TIMEOUT and ELSE parts are not protected by time limits. There is nothing within a `dialog_sequence` to prevent its "last ditch" efforts from themselves going into infinite loops. There are several reasons for this. Pragmatically, one must draw the line on timing somewhere and at some point trust code intended to compensate for timing difficulties not to introduce them itself. The fact that the programmer is writing a TIMEOUT part's statement sequence should be reminder enough that that algorithm must be cycle-free and fast. Additionally, the fact that all attempts had to have been made before the ELSE part is reached should indicate that much time has been wasted before the ELSE part begins, thus it too should be straightforward and fast. If these algorithms are such that their timing is doubtful, the `dialog_sequence` could be encased within an attempt of another such statement.

The sequences of statements following DISCUSS statements are not timed. They are for specialized post-processing for the different algorithms and are logically outside of the `dialog_sequence`'s protection, just as are those in the TIMEOUT part and within the ELSE part.

3.4.4. TIMEOUT and ELSE Parts Not Mutually Exclusive

The TIMEOUT and ELSE parts are not mutually exclusive. Each of these sections provides a last ditch opportunity for the enclosing process to achieve the `dialog_sequence`'s goal after the failure of the other attempts to achieve it. It might seem that the last ditch actions would always be the same. Their statement sequences often would be identical. However, in the TIMEOUT case, the statements are reached because one or more of the other attempts have taken too long in the effort to achieve the goal, while in the ELSE part the statements are reached because *all* of the attempts have failed for other than timing reasons in their efforts to achieve the goal.

One might consider defining the TIMEOUT part such that non-timing failure of the preceding attempt initiated its execution so that it would serve as an ELSE part as well. That would neglect the difficulties of programming the sequence of statements in the TIMEOUT part in the case in which the programmer desired to make use of the information about why the sequence of statements was reached. There could be a wealth of usable information in the knowledge of which of the cases requiring last ditch efforts had occurred. When an attempt fails, information is lost about why. Having a separate alternate algorithm for the situation of timeout as opposed to falling through an only attempt adds a little more information for programming at least the last ditch algorithm.

The ELSE part and the TIMEOUT part do not protect each other. The philosophy that they both are last ditch efforts prohibits either of them from being used as a backup algorithm for the other. If the ELSE part ''times out'', it is presumed unlikely that there will still be time for the algorithm in the TIMEOUT part to complete. Similarly for some failure of the algorithm following the TIMEOUT part being handled by the ELSE part.

3.4.5. Exceptions

An exception propagating from within a DISCUSS statement signals failure of that attempt while an exception propagating from within the sequence of statements following a DISCUSS statement, in the TIMEOUT part, or in the ELSE part signals failure of the dialog_sequence. These cases are identical to the cases of execution of FAIL statements in the same locations.

Should there be no encompassing attempt in the absence of success, the process is allowed to provide some alternate action in the form of a generic exception handler (such as for OTHERS in Ada). Should there be no encompassing attempt in the absence of success, and should the process provide no alternate action in the form of a generic exception handler, the process becomes terminated (completed, abnormal).

Except at the outermost level of a process, an explicit exception handler for `FAIL` is not allowed. The colloquy is intended to avoid the pitfalls of ad hoc recovery. The use of `OTHERS` to build a handler which can catch `FAIL` has been explicitly ruled out except at the highest level of nesting where ad hoc recovery may be necessary. The causes of a failure of this kind can be so disparate that only general restoration and possible re-try can be expected to cover them all, whereas explicitly programmed handlers can easily omit actions for important cases as well as fail to properly discriminate among the cases.

3.4.6. Nested Commitment

It might seem to be a consequence of the nested recovery rules that, at least for communications-oriented `dialog_sequences`, objects become committed or unrecoverable upon successful completion of an attempt in a nested `dialog_sequence`. This is not true. With nested `dialog_sequences`, all objects are recovered if the process backs up far enough. All communicants must have the same nesting structure. In Figure 3.7, the final value of Y is 1 rather than 3; it does not matter to which process Y actually belongs, since that process must have been a participant in dialog A as well as dialog B.

3.5. Summary

We have introduced a new linguistic construct, the colloquy, which solves the problems identified in the earlier language facilities proposed for situations requiring backward error recovery, and is as powerful as all of them. The major features that distinguish the colloquy are:

(1) The inclusion of explicit and general timing constraints. This allows processes to protect themselves against any difficulties in communication that might prevent them from meeting real-time deadlines. It also effectively deals with the problem of deserter processes. No process is required to wait forever for a process that will not enter a dialog.

```
X := 1; Y := 1;
enter A
    X := 2; Y := 2;
    enter B
        Y := 3;
    B is successful
    ( X = 2 ) & ( Y = 3 )
A fails
( X = 1 ) & ( Y = ? )
```

Event Sequence for Hypothetical Nested `dialog_sequences`

Figure 3.7

(2) The use of a two-level acceptance test. This allows much more powerful error detection because it allows the tailoring of acceptance tests to specific needs.

(3) The reversal of the order of priority of alternate communication attempts and of recovery points. This allows processes to choose the participants in their alternate algorithms rather than being required always to deal with a single set of processes.

(4) A complete and consistent syntax that is presented as extensions to Ada but could be modified and included in any suitable programming language.

The contributions of this chapter to resolving the eight issues stated at the end of Chapter 2 are listed below.

(1) The unexpected delay problem is directly addressed by the `TIMEOUT` part of the `dialog_sequence`. The syntax emphasizes that time is of utmost importance by requiring even the absence of timing constraints to be stated explicitly. Implementation overhead relative to the complexity of the semantics remains to be resolved by future experimentation.

(2) The complexity of using these constructs is controlled in several ways. The semantic rules can be enforced through static checks on the syntax. The `FAIL` statement provides easy exits without convoluted code. Separation of concerns is provided by separate tests for each alternate algorithm. This in turn encourages the tests to be more specific to each alternate. Similarly, the association of the `dialog_name` with the `dialog_function` combines the global conditions to be tested into a single place.

(3) Parallelism is basic to the colloquy, but one can use the *same* construct for sequential programming as well. The inversion of the conversation's communication-recovery relationship loosens the restrictions on the autonomy of processes. Under the colloquy, processes may use different communications partners in alternate algorithms.

(4) In that these concepts and their syntactic structures address the problems that have been identified with previous proposals, they are certainly more complete. However, as we shall see in Chapter 5, there are further problems that have not been noted in either previous proposals or the dialog and colloquy as described so far.

(5) The placement issue remains to be addressed by future experimentation with the use of these ideas.

(6) As for taking reality into account, we have already dealt with the unexpected delay problem, Chapter 4 demonstrates that there is *an* implementation of these ideas, and we will later deal with unrecoverable objects (see below). Whether backward error recovery will prove useful in real-time systems depends upon experimentation with these ideas. We are reminded that the problems of the previous proposals did not allow them to be amenable to experimentation in realistic systems.

(7) This chapter has not addressed unrecoverable objects at all. That will be addressed in Chapter 6.

(8) The latency issue is not addressed further in this work other than weakly by the provision for multiple, hence more specific, acceptance tests.

The dialog and colloquy are broad generalizations of previous software fault-tolerance proposals. This view puts what we had called those proposals' shortcomings into a better light. Now, rather than shortcomings, they are the restrictions which make those proposals special cases of this more general concept. That those proposals are special cases of the colloquy is demonstrated by example in Appendix B. The fact that we had termed those restrictions "shortcomings" indicates that we still consider them ill-advised. However, the illustration syntax for the `DISCUSS` statement and `dialog_sequence` would force such restrictions to be deliberate decisions on the part of the programmer/designer.

CHAPTER 4

Implementability

This chapter demonstrates that the `DISCUSS` statement and the `dialog_sequence` can be implemented. The implementation described here is probably not optimal.

An existing LALR(1) grammar for Ada, amended by BNF rules equivalent to those given in Appendix A, remains at least a LALR(1) grammar. Hence an unambiguous parse of a `dialog_sequence` and its constituent `discuss` statements is possible. Therefore, a compiler can generate code or generate calls to support code implementing the algorithms presented below at the appropriate boundaries within the `dialog_sequence`. This chapter describes the code and data structures that must be generated and the support routines that must be invoked at those boundaries. For generality, we show via pseudocode the functionality required of these routines. We discuss issues arising from the definitions of dialog, `DISCUSS` statement, colloquy, discussion, and `dialog_sequence` and how an implementation might address them.

First, we discuss some of the underlying mechanisms that we assume exist in some form in an implementation of Ada whether backward error recovery is included or not. Assuming these mechanisms eliminates some of the detail. Next, we describe extensions of assumed existing data structures of an Ada implementation and some new data structures used in this extension to the implementation for backward error recovery. The bulk of the presentation involves descriptions of additional algorithms employed by the language support system and how they relate to the program source and to other parts of the support system.

4.1. Underlying Mechanisms

For brevity, we make certain assumptions about the language support system to which additional facilities supporting backward error recovery are to be tied. These involve a recovery cache mechanism, the data structures for providing the illusion of parallel processes, the data structures and mechanism implementing Ada DELAYS, the mechanism for propagating exceptions, the mechanism for queuing messages related to the Ada rendezvous, the existence of certain word-locking algorithms, and the implementability of implicit operations during branching operations.

We assume the language support system's or hardware's ability to perform the caching functions on demand. Lee [Lee78] and Lee, et al [Lee80] discuss this. They concentrate on single-process systems and state that it is easy to extend to multiple processes which make independent caching demands.

We assume the support system keeps track of the information necessary for realizing a process in a table which we call a *process control block*. We will describe additional fields of this table for information specific to an individual process as it deals with others through dialogs.

The basic facilities for timers occur often in operating systems. Usually there is only a single hardware device (per processor) interrupting the processor at regular intervals, providing a basic granularity of time measurement. We assume a queue of "events" is kept and compared against the cumulative "ticks" of the interrupting device. An implementation of Ada must include a similar facility. Ada allows individual processes to specify scheduling DELAYS and a form of timeout on individual communications (messages). For implementing the `dialog_sequence`, we create a new kind of timer queue entry and define how the entry is to be associated with the `dialog_sequence` of a particular process.

In Ada, there are two possible exits from any statement or block. One of these is to be taken if the statement succeeds. The normal instruction sequencing mechanism is used for this path. The other is to be taken if an exception is raised in or propagated to the statement. When an exception is raised, the address of the currently appropriate handler code (or the code to discriminate among handlers) must be available from some standard place different than that implied by the normal instruction sequencing mechanism. For exception propagation, an alternate return address must be available similarly. The nesting of exception handler contexts (`declare` blocks) implies that this address is also stackable.

In Ada, the message protocol is such that the sender is suspended until the message is received and a reply returned (beginning and end of rendezvous). The messages exist on the receiver's "entry queues". Although suspended, an indication exists in the sender as to which process is the receiver. This is to facilitate retraction of the message under such eventualities as expiration of DELAYS and ABORTS targeting the sender. Also, the number and absolute names of the receiver's queues are subject to change due to "entry families". Since the index type of an entry family can cover many unused values, it is impractical for an implementation to provide a static data structure for the maximum possible number of entries. We therefore assume a dynamic number of data structures representing entry queues and that associated dynamic queue names are checked at an early stage of the message acceptance process.

We also assume the existence of certain word-locking algorithms. Under such algorithms, the spaces allocated to individual variables can be locked so as to restrict access to a single process or a limited group of processes. We further specify that the desired algorithms permit the access rights to be temporarily restricted to a subset of the already-privileged group of processes.

Pausing during a branching operation for another, implicit operation is not new. It is essentially what is required in Ada for branches (or RETURNS) out of blocks (procedures) which declare tasks. The branches in those cases cannot complete until the declared tasks terminate. Implementations of Ada exist, so this kind of branch can be implemented.

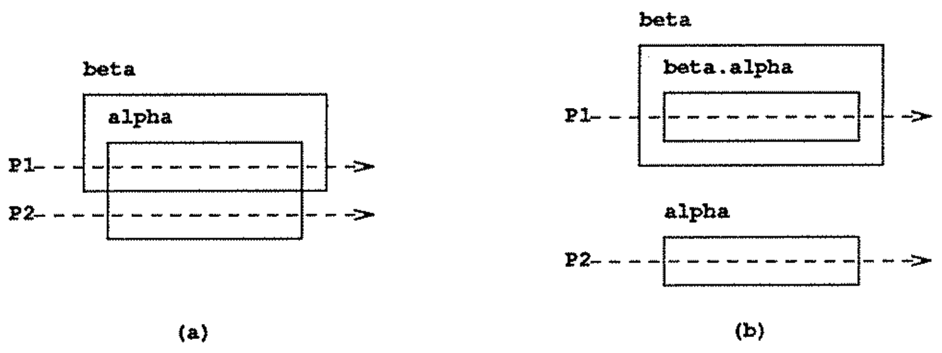
4.2. Abstract Data Structure

This section describes the extensions to an Ada implementation's data structures and some additional data structures that are useful in implementing the dialog_sequence and DISCUSS statement. First, there is some discussion of the translation from source to internal naming. This provides a basis for understanding of the organization of some of the information in the data structures. Then, the primary data structure for implementing dialogs is presented. The descriptions of extensions to the process control block and timer queue complete the data structures section.

4.2.1. Internal Dialog Naming

As Chapter 3 states, proper nesting of DISCUSS statements, the syntactic evidence of dialogs, is ensured by the syntax itself. Unfortunately, that only holds for single processes. Suppose processes P1 and P2 each enter DISCUSS statements with dialog_name alpha. Suppose further that P1's DISCUSS statement referencing alpha is nested within a DISCUSS statement referencing dialog_name beta, but P2's is not. Syntactically, everything is fine, but the situation is as depicted in Figure 4.1a. Communication between P1 and P2 within alpha presents an obvious case of smuggling as regards beta. Since this syntactic situation can legitimately occur, we allow it, but insist that the dialog of which P2 is a member is actually (internally) named alpha, whereas P1 is a member of an entirely different dialog actually named beta.alpha as depicted in Figure 4.1b.

The internal names of dialogs referenced in procedure bodies are formed by appending the dialog_names mentioned in the procedure bodies to the internal dialog names applicable at each point of call for the procedure. For instance (Figure 4.2), suppose a procedure P contains a dialog_sequence with DISCUSS statements referencing dialog_names alpha and beta. Further suppose that P can be called from within some DISCUSS statement referencing dialog_name gamma and from outside any DISCUSS statement. We have five potential unique internal dialog names: alpha, beta, gamma,



Improperly and Properly Nested Dialogs

Figure 4.1

```
PROCEDURE p is ... begin ...       [name] ≡ dialog_entry for  dialog "name"
   DISCUSS alpha BY ...
 OR DISCUSS beta BY ...
END p;                           set of outermost dialogs
...                              ...........
SELECT                           . [alpha] .
   DISCUSS gamma BY ...          .         .   set of nested dialogs
      p; ...                     . [beta]  .   .................
END SELECT;                      .         .   . [gamma.alpha] .
...                              . [gamma]-----.               .
 p;                              ...........   . [gamma.beta]  .
...                                            .................
```

Internal Names From Call Chain

Figure 4.2

`gamma.alpha`, and `gamma.beta`, resulting in five of the `dialog_entry`s that we describe below. If an implementation allowed unbounded recursion in procedure calls, `dialog_entry`s would have to be built at execution time rather than at compile time. Including the additional complexity required of the data structures and algorithms below in order to handle unbounded recursion would not be instructive. For this presentation, we assume statically determinable bounds on recursion.

4.2.2. Information Specific to Each Dialog

The implementation maintains a `dialog_entry` for each dialog instance (Figure 4.3). There is a collection of `dialog_entry`s for the dialogs at the outermost level. Besides its name and where to find the entry point for the global acceptance test function, each dialog has a dialog (`parent`) within which it is nested, and a collection of dialogs (`children`) nested within it. The shared variable list is represented with information necessary for finding them, the entry points of the code specific to this list of shared variables for saving, restoring from, and discarding a recovery point for them, and information for locating that recovery point in whatever recovery cache mechanism is being used. This last item might be the value returned as a *cachemark* from the `save` instruction [Lee80]. There are a lock on the `dialog_entry` itself to prevent the addition of new participants during evaluation of the global test and an `active` flag to aid in starting a new dialog and in limiting searches of the collection of `dialog_entry`s. If

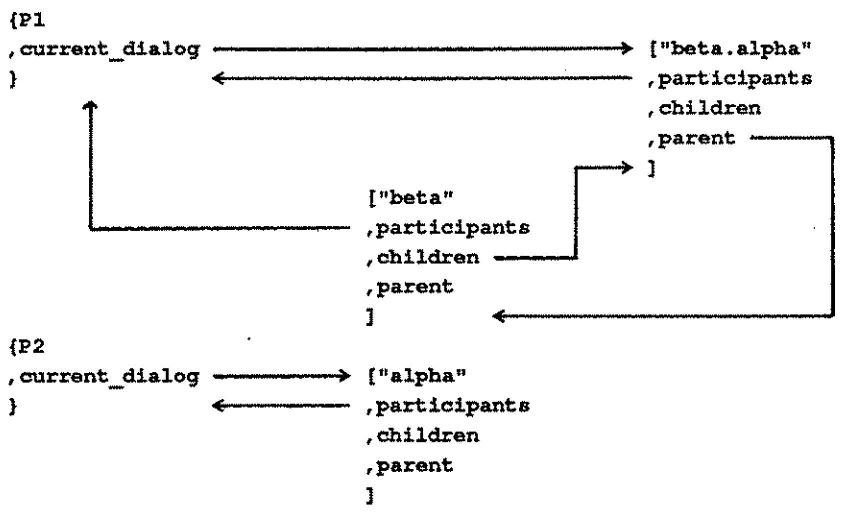
```
dialogs_table ->
        set_of(dialog_entry)            -- static structure

dialog_entry  ->
        [name                         :string         -- static unless noted
        ,function_code                :code_address
        ,parent                       :reference_to(dialog_entry)
        ,children                     :set_of(dialog_entry)
        ,shared_variable_addresses                    -- static structure
        ,shared_variable_save_code    :code_address
        ,shared_variable_restore_code:code_address
        ,shared_variable_discard_code:code_address
        ,shared_variable_cachemark                    -- dynamic value
        ,lock                         :boolean        -- dynamic value
        ,active                       :boolean        -- dynamic value
        ,participants                 :set_of(reference_to(process_control_block))
                                                      -- dynamic structure
        ,locked_out                   :set_of(reference_to(process_control_block))
                                                      -- dynamic structure
        ]
```

Sample Implementation Data Structure

Figure 4.3

a dialog is not active, i.e. has no participants, then neither are the dialogs nested within it. If the `active` flag indicates a process has not entered, the next one to do so must be the first and can establish a recovery point for the shared variables. Each process participating in a dialog can be found through the `participants` set. Similarly, the `dialog_entry`(s) of the dialog(s) of which any process is a member can be found through that process' control block (Figure 4.4). Finally, as the current dialog instance ends, the set (`locked_out`) of processes locked out of the dialog during evaluation of the global test can be found and allowed to create a new instance.

The static nesting of DISCUSS statements within each process, and hence the full dotted names of all potential dialogs is evident in the program text. Much of the data structure contains static information, so can be built at compile time rather than being reconstructed at execution time as dialog instances come and go. The `active` flag causes the `dialog_entry` to be ignored until a process actually tries to enter a dialog of that name. The `lock` flag prevents new participants from entering during evaluation of the



Process Control Block / `dialog_entry` Linkage

Figure 4.4

global test. Parts of the data structure do have to be filled in at execution-time as dialog instances occur. One such item is the reference path from a `dialog_entry` to the participating processes as they enter and exit an instance of the dialog. The potential maximum size of the participant collection could also be pre-computed.

4.2.3. Process Control Block

Additional information in a process' control block due to the inclusion of `dialog_sequences` is reviewed in Figure 4.5. We use ''`next_instruction`'' to refer to the stored program counter when a process is descheduled. The failure and success continuations are the target locations for the completion of dialogs, appropriating the exception/alternate return address mechanism assumed earlier. Roughly, these continuations are the beginning of the next attempt and the statement sequence following the DISCUSS statement, respectively. For the last attempt, the failure continuation is the beginning of the ELSE part. There is a record (`end_of_sequence`) of whether the process is currently executing the last

```
process_control_block ->
        [universal_name
        ,next_instruction     :code_address
            ...
        ,current_dialog       :reference_to(dialog_entry)
        ,failure_continuation:stack_of(code_address)
        ,success_continuation:stack_of(code_address)
        ,test_address         :stack_of(code_address)
        ,end_of_sequence      :stack_of(boolean)
        ,branch_stack         :stack_of(code_address)
        ,recovery_point_info  :stack_of(mechanism_dependent)
        ,accepted             :boolean
        ,dialog_cleaned_up    :boolean
            ...
        ]
```

Additions to Process Control Block

Figure 4.5

dialog in the `dialog_sequence` at each level of nesting. The address of the local acceptance test for the current dialog is available for use in branches. This is augmented by a stack (`branch_stack`) of alternate success continuations for branches across dialog boundaries. These values are stacked and popped as the process enters and exits nested dialogs, as is the recovery point information. If recovery points are implemented entirely via hardware, there may be very little information to keep in this last stack. The `current_dialog` field references the `dialog_entry` of the innermost dialog of which the process is a member. The `accepted` flag denotes whether the process has or has not yet evaluated its local acceptance test `TRUE` in the current dialog. The `dialog_cleaned_up` flag is an artifact of the centralized algorithms we present below. The process that evaluates the global test pops the continuation stacks for the other participants yet they continue at the beginning of the same algorithm as it did; hence, they use the flag to branch around the algorithm.

4.2.4. Timer Queue

A timer queue entry (Figure 4.6) is constructed after evaluation of the timing constraint expression, if any, and before the process performs the dialog entry algorithm for the first dialog in a timed `dialog_sequence`. The `kind` field distinguishes dialog `TIMEOUT` entries from other events on the timer queue. The entry contains the expiration time, a reference to the process, and a means of associating it

```
timeout_entry  ->
        [expiration_time    :time
        ,kind               :( ... ,TIMEOUT)
        ,the_process        :reference_to(process_control_block)
        ,surrounding_dialog:reference_to(dialog_entry)
        ,timeout_part       :code_address
        ]
```

Timer Queue Entry for `TIMEOUT`

Figure 4.6

with the `dialog_sequence` within the process. For this association, we use a reference to the immediately enclosing dialog, where `null` indicates a timing constraint on the outermost `dialog_sequence`. The entry point of the `TIMEOUT` part is also included in the timer queue entry.

4.3. Operation of Dialogs: Dialog Algorithms

The implementation manages the bookkeeping for membership in dialogs, communication without smuggling, executing local and global tests, branching based on success or failure, and timeouts.

Figure 4.7 is the form of the source text of a `dialog_sequence`. It will be used as a running example. Figure 4.8 shows placement of the "calls" to many of the algorithms described in this section relative to other code generated for the `dialog_sequence` of the running example. For reference, the labels in the two figures correspond. We use the Ada named-parameter association syntax. The `TIMER_INTERVAL` parameter to algorithm `PRIMARY` and the `LOCAL_TEST` parameter to algorithm

```
L01: SELECT
L02:   DISCUSS alpha BY
L03:     statement_seq_1
L04:   TO ARRANGE beta;
L05:   statement_seq_2
L06:  OR
L07:   DISCUSS gamma BY
L08:     statement_seq_3
L09:   TO ARRANGE delta;
L10:   statement_seq_4
L11:  TIMEOUT interval_1
L12:   statement_seq_5
L13:  ELSE
L14:   statement_seq_6
L15: END SELECT;
```

Example Source Text Form

Figure 4.7

```
L01:   PRIMARY(TIMER_INTERVAL => interval_1, TIMEOUT_PART    => L12
              ,DIALOG_NAME    => alpha     , COMMIT_LOCATION => L05
              ,FAIL_LOCATION  => L07       , TEST_LOCATION   => L04
              ,END_OF_SEQUENCE=> FALSE                             )
       statement_seq_1
L04:   ACCEPTANCE_TESTS(LOCAL_TEST => beta)
L05:   COMMIT
       statement_seq_2
       goto L15
L07:   ALTERNATE(DIALOG_NAME    => gamma, COMMIT_LOCATION => L10
                ,FAIL_LOCATION  => L14  , TEST_LOCATION   => L09
                ,END_OF_SEQUENCE=> TRUE                        )
       statement_seq_3
L09:   ACCEPTANCE_TESTS(LOCAL_TEST => delta)
L10:   COMMIT
       statement_seq_4
       goto L15
L12:   LAST_DITCH
       statement_seq_5
       goto L15
L14:   LAST_DITCH
       statement_seq_6
L15:   ;
```

Placement of Calls to Dialog Algorithms

Figure 4.8

ACCEPTANCE_TESTS are passed "by name" [Inge61] if the implementation does not generate the code for the algorithms in-line.

Briefly, algorithm PRIMARY first establishes the timer queue event for the dialog_sequence. If the process has not begun executing one of algorithms COMMIT and LAST_DITCH after TIMER_INTERVAL time units, the process' then current dialog will fail and the process will continue at the TIMEOUT_PART. Algorithm PRIMARY next makes the process a participant in the named dialog with the provisions that on success the process will continue at the COMMIT_LOCATION (algorithm COMMIT) and that on failure it will continue at the FAIL_LOCATION (algorithm ALTERNATE). After this, the process proceeds to execute the body (statement_seq_1) of the first DISCUSS statement.

Upon completion of the DISCUSS statement's body, algorithm ACCEPTANCE_TESTS evaluates the process' LOCAL_TEST and, if necessary, the dialog's global test. Based on the result and the values passed to algorithm PRIMARY, algorithm ACCEPTANCE_TESTS then branches to one of algorithms COMMIT and ALTERNATE.

Algorithm COMMIT cancels any timer events on the dialog_sequence and releases each participant of the successful dialog to continue at the sequence of statements following its DISCUSS statement. In case there was a branch from the DISCUSS statement's body to a destination outside that body, the TEST_LOCATION passed to algorithm PRIMARY will have been used, and algorithm COMMIT will effect the final branch to the intended destination. Algorithm ALTERNATE, like algorithm PRIMARY, makes the process a member of the named dialog with the same respective provisions. Unlike algorithm PRIMARY, it only deals with timer events when it cancels any events on dialog_sequences nested within the dialog whose failure has led the process to algorithm ALTERNATE. As well as canceling these timer events, the algorithm releases the participants of the failed dialog, but in a failed mode as opposed to the successful mode employed by algorithm COMMIT. Were there a third attempt in the example dialog_sequence, the address passed to the first instance of algorithm ALTERNATE would be that of the instance naming the dialog desired for the third DISCUSS statement.

Algorithm LAST_DITCH, like algorithm ALTERNATE, cancels timer events on nested dialog_sequences and releases participants of the failing dialog. Like algorithm COMMIT, it also cancels any timer event on this dialog_sequence, before allowing the process to continue at the appropriate last-ditch try in the dialog_sequence.

The disparities in processing of the timer events and recovery points are the major complications differentiating the dialog algorithms. A timer event on the dialog_sequence is established by algorithm PRIMARY, and canceled by one of algorithms COMMIT and LAST_DITCH. It is not touched between dialogs in this dialog_sequence by algorithm ALTERNATE. The recovery point local to this process is saved by algorithm PRIMARY, used for recovery by algorithms ALTERNATE and LAST_DITCH, and discarded by algorithms COMMIT and LAST_DITCH. The recovery point for any data shared by a dialog is saved by algorithm PRIMARY or ALTERNATE, used for recovery by algorithms ALTERNATE and LAST_DITCH, and discarded by algorithms ALTERNATE, COMMIT, and LAST_DITCH. The other complication lies in the special handling of continuations in algorithm COMMIT for branches out of a DISCUSS statement.

We describe the algorithms roughly in the order they occur in the example, PRIMARY, ACCEPTANCE_TESTS, COMMIT, ALTERNATE, and LAST_DITCH. The areas of commonality among these algorithms have been factored out to "macros" called ENTER_DIALOG, CANCEL_TIMERS, and EXTRICATE_PARTICIPANTS. Their descriptions are interspersed among those of the algorithms. Finally, we describe several utility algorithms, TIMEOUT, IMPLICIT_OTHERS, and BRANCH, that help explain how the algorithms above mesh with other parts of the language support system.

4.3.1. Algorithm PRIMARY

Algorithm PRIMARY (Figure 4.9) is really very simple. The interval specified in the TIMEOUT part of the `dialog_sequence` is evaluated and a timing event installed on the timer queue assumed above. Then, a recovery point is saved for this process and the process enters the named dialog.

As for notation in these algorithms, we assume the existence of certain obvious routines, represented here by `now` (returning the current time in whatever units), `install_timer_queue_entry` which must exist in some form in an Ada implementation anyway, and `push`, `pop`, and `top` for manipulating stacks. Constants and variables local to the algorithms are in capitals, and calls for expansion of macros and conditional "compilation" are bracketed by `<<< ... >>>`. We use `self` to reference the currently executing process' control block. Figure 4.9 line 7 prevents the process from being extracted from a dialog by algorithm LAST_DITCH, since it would not have entered a dialog before the interval expired. The pseudocode beyond Figure 4.9 line 9 is assumed not to be interruptible from the scheduling standpoint or by the other algorithms.

```
Algorithm PRIMARY(TIMER_INTERVAL :THUNK duration;TIMEOUT_PART    :code_address
                 ;DIALOG_NAME    :string        ;COMMIT_LOCATION:code_address
                 ;FAIL_LOCATION  :code_address  ;TEST_LOCATION  :code_address
                 ;END_OF_SEQUENCE:boolean                                    ) ≡
    END_TIME  := now + TIMER_INTERVAL
    if END_TIME ≤ now then
        self.dialog_cleaned_up := TRUE
        goto TIMEOUT_PART
    end if
    install_timer_queue_entry(expiration_time    => END_TIME
                             ,kind               => TIMEOUT
                             ,the_process        => self
                             ,surrounding_dialog => self.current_dialog
                             ,timeout_part       => TIMEOUT_PART)
    push(self.recovery_point_info,save_recovery_point_for_process(self))
    <<<ENTER_DIALOG>>>
end Algorithm PRIMARY
```

Algorithm PRIMARY

Figure 4.9

4.3.1.1. Macro ENTER_DIALOG

Because ENTER_DIALOG (Figure 4.10) is a macro, expanded in-line, the local variables and formal parameters of its calling algorithms carry over. When entering a dialog, a process saves the addresses that will be used when it exits, notes whether or not the dialog corresponds to the last DISCUSS statement in the `dialog_sequence`, flags itself as not having passed its local acceptance test for the dialog, and references the `dialog_entry` appropriate to the desired dialog. That entry is the one with the `dialog_name` used in the DISCUSS statement and nested within the current dialog (Figure 4.10 line 11). If there is no current dialog, the choice is among the roots of the trees (formed by the `parent` and `children` relationships) in the forest formed by the outermost dialogs (line 8). Now that the process is ready to enter the appropriate dialog, it may become suspended while a pre-existing instance of the dialog evaluates its global acceptance test and its participants exit. Line 15 allows the suspended processes to be found easily for revival as soon as the old instance becomes inactive. Since line 16 is essentially a scheduler request, the algorithm employing the macro is "interruptible" at that point as we have been

```
Macro ENTER_DIALOG ≡
    push(self.failure_continuation,FAIL_LOCATION)
    push(self.success_continuation,COMMIT_LOCATION)
    push(self.test_address        ,TEST_LOCATION)
    push(self.end_of_sequence     ,END_OF_SEQUENCE)
    self.accepted := FALSE
    if self.current_dialog = NULL then
        D := x where (x ∈ dialogs_table
                         | x.name = DIALOG_NAME)
    else
        D := x where (x ∈ self.current_dialog.children
                         | x.name = DIALOG_NAME)
    end if
    if D.lock then
        D.locked_out := D.locked_out ∪ {self}
        hang
    end if
    if ¬ D.active then
        execute(D.shared_variable_save_code)
        D.active := TRUE
        D.lock   := FALSE
    end if
    self.current_dialog := D
    D.participants := D.participants ∪ {self}
end Macro ENTER_DIALOG
```

Macro ENTER_DIALOG

Figure 4.10

using the term.

The set of variables in the SHARES list associated with a dialog are cached for recovery. This is done under control of the support code rather than in-line instructions of participating processes. Which process will be first to enter a given dialog instance is not known a priori. Code cannot be generated in-line for a particular process to establish the recovery point. The compiler generates code (`shared_variable_save_code` in the `dialog_entry`) to be called on behalf of the first process which enters the dialog to save the recovery point. Code is also generated to restore or to discard the shared variables' cached values (`shared_variable_restore_code`, and `shared_variable_discard_code` in the `dialog_entry`). This code may use tables (`shared_variable_addresses` in the `dialog_entry`) to help locate the variables. The set of variables shared within any dialog is of course known at compile time.

If this is the first process to enter this instance of the dialog (line 18), a recovery point is saved for the variables in the `SHARES` list. Finally, the process links its control block with the `dialog_entry` as described in Figure 4.4 and proceeds to execute the body of its `DISCUSS` statement.

4.3.2. Communication During Dialogs

A major function of dialogs is to provide backward error recovery for communications between processes.

To prevent smuggling via messages, the implementation attaches to each message the internal dialog name for the dialog of which the sender is a member. A receiving process is not allowed any information about the existence of the message until the receiver becomes a participant in the same dialog. If the dialog ends before the message is received, the message is removed as part of the normal clean-up for the dialog. In that case, the suspension of the sender in Ada implies that the sender would not have proceeded to its acceptance test, so the dialog ending without message receipt would be due to expiration of a timing constraint or failure of a different process in the sender's dialog. Messages which disappear due to dialog failure never actually existed — this is consistent with the idea of a dialog being an atomic action.

Internally, the receiver has several entry queues for each source entry queue name. The queues' full names are the source names with internal dialog names appended. For example, suppose the `accept` is within two nested dialogs `ALPHA` and `BETA`. The presence in the source of `accept e` tells the implementation to actually generate `accept ALPHA.BETA.e`.

4.3.3. Algorithm ACCEPTANCE_TESTS

Having completed execution of the DISCUSS statement's body, the process begins algorithm ACCEPTANCE_TESTS (Figure 4.11). The local acceptance test is evaluated. If that test does not fail, the process becomes non-interruptible after Figure 4.11 line 4 as described for algorithm PRIMARY. The process marks itself as having passed its local test in case it has to wait for others to do so. Most of the algorithm involves deciding whether the process should go ahead with evaluating the global acceptance test. It should if (Figure 4.11 lines 6 to 10) there remain no participants that have not passed their local tests. Line 7 deals with members of a nested dialog that are awaiting evaluation of its global test. Once the decision has been made to execute the global test, no new processes may enter the dialog. These algorithms do not provide for it. If this is the last process to arrive at the tests, the dialog is locked to prevent the entry of more participants until this instance of the dialog ends. The global test (line 12) is

```
 1 Algorithm ACCEPTANCE_TESTS(LOCAL_TEST:THUNK boolean) ≡
 2     if ¬ LOCAL_TEST then
 3         goto top(self.failure_continuation)
 4     else
 5         self.accepted := TRUE
 6         if {i ∈ self.current_dialog.participants |
 7                (i.current_dialog ≠ self.current_dialog)
 8                   or
 9                (¬ i.accepted)}
10             = ∅ then
11                self.current_dialog.lock := TRUE
12                if ¬ self.current_dialog.function_code then
13                    goto top(self.failure_continuation)
14                end if
15                -- will drop through to COMMIT
16         else
17             hang -- will proceed using success_ or failure_continuation
18         end if
19     end if
20 end Algorithm ACCEPTANCE_TESTS
```

Algorithm ACCEPTANCE_TESTS

Figure 4.11

evaluated in interruptible mode after which the process will have detected failure of the dialog or drop into algorithm COMMIT for success. The processes that were not the last to pass their local tests become suspended (line 17), waiting for the last process to determine success or failure of the dialog and where they should resume execution.

It is because the tests are unpredictably long and may raise exceptions that the process is interruptible while evaluating them. This allows algorithm TIMEOUT or the exception handling mechanism to take over if need be.

4.3.4. Algorithm COMMIT

A process arrives at algorithm COMMIT (Figure 4.12) via one of two routes. It may evaluate the dialog's global test TRUE and fall through from algorithm ACCEPTANCE_TESTS. Alternatively, it may have remained suspended in algorithm ACCEPTANCE_TESTS while another process took the first route. In the latter case, the other process would have done all that was necessary to release this process from the dialog, and would therefore have set this process' `dialog_cleaned_up` flag. The entirety of this

```
Algorithm COMMIT ≡
    if self.dialog_cleaned_up then
        self.dialog_cleaned_up := FALSE
    else
        <<<CANCEL_TIMERS(COMMITTING)>>>
        <<<EXTRICATE_PARTICIPANTS(COMMITTING)>>>
        if ¬ stack_is_empty(self.branch_stack) then
            goto pop(self.branch_stack)
        end if
    end if
end Algorithm COMMIT
```

Algorithm COMMIT

Figure 4.12

algorithm is non-interruptible.

In the former case, this process holds the responsibility for releasing all participants of the dialog from it and from the dialog_sequences of which it is a part. In the expansions of macros CANCEL_TIMERS and EXTRICATE_PARTICIPANTS, definition of the pseudo-variable COMMITTING enables the special processing needed to exit the entire dialog_sequence. Because the processes are leaving their respective dialog_sequences, any timer queue entries established for those dialog_sequences will be removed as well as entries for dialog_sequences nested within them (because this dialog was successful, there are no nested dialog_sequences). Likewise with macro EXTRICATE_PARTICIPANTS, COMMITTING causes the recovery points for the participant processes to be discarded. That the participants of the dialog are COMMITTING to its effects prohibits recovery from those recovery points or from the recovery points for variables in the SHARES list. COMMITTING also provides the use of the success_continuation over the failure_continuation when this process releases the other participants from their suspension due to algorithm ACCEPTANCE_TESTS.

Finally, if the process is exiting the dialog as part of a non-local branch, it continues on the path of the branch rather than falling through to the statements following the DISCUSS statement in the attempt. The description of algorithm BRANCH contains more explanation of the non-local branching mechanism.

4.3.4.1. Macro CANCEL_TIMERS

As a process exits the last dialog in a discussion[†], any timer event on the dialog_sequence must be canceled. Failure of a dialog includes termination of any dialog_sequences active within it at the time. Hence, each of algorithms COMMIT, ALTERNATE, and LAST_DITCH uses macro CANCEL_TIMERS (Figure 4.13). Recall that the dialog_entrys are organized as a forest with nesting indicated by

[†]Recall that a discussion is the sequence of dialogs that the process actually enters at execution time in correspondence to a dialog_sequence. If one of the dialogs succeeds, the process does not enter the dialogs indicated by the remaining DISCUSS statements. If all of the dialogs fail, the timer event on the dialog_sequence must still be canceled.

```
Macro CANCEL_TIMERS ≡
    D := self.current_dialog
    SUBJECT_SET := {dialog d | (d = D) or (d.active and
                                          d.parent ∈ SUBJECT_SET)}
    TMP_QUEUE := new <timer queue>
    while ¬ queue_is_empty(timer_queue) loop
        TIME_ENTRY := dequeue(timer_queue)
        if       TIME_ENTRY.kind ≠ TIMEOUT
            or else
                 TIME_ENTRY.the_process ∉ D.participants
            or else
                 TIME_ENTRY.surrounding_dialog ∉ (SUBJECT_SET ∪ {D.parent})
        then
            enqueue(TMP_QUEUE,TIME_ENTRY)
<<<IFUNDEF COMMITTING>>>
        else if TIME_ENTRY.surrounding_dialog = D.parent
        then
            TMP_STACK := new <stack of boolean>
            TMP_DIALOG:= TIME_ENTRY.the_process.current_dialog
            while TMP_DIALOG ≠ D loop
                TMP_DIALOG := TMP_DIALOG.parent
                push(TMP_STACK,pop(TIME_ENTRY.the_process.end_of_sequence))
            end loop
            if ¬ top(TIME_ENTRY.the_process.end_of_sequence)
            then
                enqueue(TMP_QUEUE,TIME_ENTRY)
            end if
            while ¬ stack_is_empty(TMP_STACK) loop
                push(TIME_ENTRY.the_process.end_of_sequence,pop(TMP_STACK))
            end loop
<<<ENDIFUNDEF>>>
        end if
    end loop
    timer_queue := TMP_QUEUE
end Macro CANCEL_TIMERS
```

Macro CANCEL_TIMERS

Figure 4.13

branching of the trees. Recall also that a dialog can fail while some of its participants are engaged in a nested dialog; for example, a process in the outer dialog may reach and fail its local acceptance test. The SUBJECT_SET consists of the dialogs that are being terminated. Under all circumstances, the timer events on the `dialog_sequence`s naming the nested dialogs are to be canceled. For processes that are leaving their current `dialog_sequence`s, the timer events on those `dialog_sequence`s are to be canceled too.

The canceling technique is pedestrian. The timer queue is copied onto a temporary queue one event item at a time. Those items corresponding to events we wish to cancel are filtered out in the process. Then, the timer queue is replaced by the filtered version.

The filter is in two stages. On lines 8 through 14, the filter can only remove `TIMEOUT` timer queue entries (line 8) established for processes that are participants (line 10) in the outermost terminating dialog (this includes participants in the nested dialogs). It will not remove entries associated with `dialog_sequence`s within which the terminating dialog's `dialog_sequence`s are nested (line 12). If the terminating dialog is `COMMITTING`, all events selected by the first stage are removed. Otherwise, the second stage (lines 16 through 30) preserves (line 26) events established for `dialog_sequence`s for which the outermost terminating dialog is an attempt (line 16) but for which it is not the last attempt in their sequences of dialogs (line 24). The loop of line 20 is used to find those `dialog_sequence`s' nesting level on the `end_of_sequence` stacks, and the loop of line 28 restores those stacks.

4.3.4.2. Macro `EXTRICATE_PARTICIPANTS`

Once a decision has been made as to the success of a dialog, the individual processes are released to make another attempt or to leave the `dialog_sequence`, and the `dialog_entry` is cleared for another instance. This is the purpose of macro `EXTRICATE_PARTICIPANTS` (Figure 4.14).

First, the process control block for each of the processes that are `participants` in the dialog is altered to indicate that the process is no longer in the dialog (lines 2 to 32). Then the dialog instance is terminated as the `dialog_entry` is reset to represent an inactive dialog (lines 33 to 43).

Comments bracketed by `--<` and `>` should be replaced by code to perform the operations the comments describe. The reader is assumed to know how to implement these operations. The participant processes which may have been marked `accepted`, and thus suspended, get marked runnable (line 3) at their `success_` or `failure_continuation`s, as appropriate (lines 6 through 9). The other continuation

```
Macro EXTRICATE_PARTICIPANTS ≡
  ∀ P ∈ D.participants loop
    --<if P hung for any reason, make it runnable>
    --< and recall any messages it may have sent >
    TMP_FAIL := pop(P.test_address)
    TMP_FAIL := pop(P.failure_continuation)
    TMP_NEXT := pop(P.success_continuation)
<<<IFUNDEF COMMITTING>>>
    TMP_NEXT := TMP_FAIL
    P.branch_stack   := <null stack>
    restore_recovery_point_for_process(P,top(P.recovery_point_info))
    if pop(P.end_of_sequence) then
<<<ENDIFUNDEF>>>
      discard_recovery_point_for_process(P,pop(P.recovery_point_info))
<<<IFUNDEF COMMITTING>>>
    end if
<<<ENDIFUNDEF>>>
    if P ≠ self then
      P.dialog_cleaned_up := TRUE
<<<IFDEF COMMITTING>>>
      if ¬ stack_is_empty(P.branch_stack) then
        P.next_instruction := pop(P.branch_stack)
      else
<<<ENDIFDEF>>>
       P.next_instruction := TMP_NEXT --WARNING check out RETURNs
<<<IFDEF COMMITTING>>>
      end if
<<<ENDIFDEF>>>
    end if
    P.current_dialog := D.parent
    P.accepted       := FALSE
  end loop
<<<IFUNDEF COMMITTING>>>
  execute(D.shared_variable_restore_code)
<<<ENDIFUNDEF>>>
  execute(D.shared_variable_discard_code)
  D.participants  := ∅
  D.active        := FALSE
  D.lock          := FALSE
  ∀ P ∈ D.locked_out loop
    --<make P runnable>
  end loop
  D.locked_out  := ∅
end Macro EXTRICATE_PARTICIPANTS
```

Macro EXTRICATE_PARTICIPANTS

Figure 4.14

and the `test_address` are thrown away. Whether the executing process is COMMITTING determines this for all participants, although the value popped is not used until line 25.

If the dialog failed, any decision to branch out of it is revoked by nullifying the `branch_stack` (line 10), and the process' recovery point is restored (line 11). The recovery point for the variables in the dialog's SHARES list is also restored upon failure (line 34). Even if the dialog is not COMMITTING, if the process has also reached the end of its `dialog_sequence`, its local recovery point is discarded as well (line 14).

The `if` of line 18 ensures that the processes, when next scheduled, will begin executing at the proper location. The process executing the algorithm embodying this macro will, of course, either fall through to the appropriate follow-on code or use the `branch_stack` as seen in algorithm COMMIT. Since that one is extricating the other participants from the dialog for them, they need not through the dialog-closing algorithms. They are so notified via line 19. If the dialog failed, line 25 ensures that the other processes will use the `failure_continuation` popped on line 6 and saved on line 9. Otherwise, there is a choice between having line 25 use the `success_continuation` popped on 7 and throwing the `success_continuation` away in favor of a value from the `branch_stack`. As long as there is a value on the `branch_stack` (line 21), the destination of a branch out of a DISCUSS statement has not yet been reached, and that value represents where the process is to resume.

Redirecting the process' `current_dialog` indication (line 30) and noting that it has not yet passed the local acceptance test of the parent dialog (line 31) completes the process' dealings with this dialog instance.

The shared variable recovery point is discarded on termination of this instance of the dialog whether successful or not (line 36), since there is no guarantee that any of the participants in the dialog will enter a dialog with the same set of shared variables as the next attempt in its `dialog_sequence`.

Without participants (line 37 after the loop lines 2 through 32), the dialog instance becomes inactive (line 38), allowing those processes that were barred entry during evaluation of the global acceptance test to proceed with creating a new instance (line 41).

4.3.5. Algorithm ALTERNATE

Algorithm ALTERNATE (Figure 4.15) terminates a dialog and begins another in a `dialog_sequence`. The initial `if` test, as in algorithm COMMIT, prevents duplication of effort in case another process terminated the dialog while this process was suspended in algorithm ACCEPTANCE_TESTS. The special processing for COMMITTING to the effects of the dialog and for leaving this `dialog_sequence` are not needed when expanding macros CANCEL_TIMERS and EXTRICATE_PARTICIPANTS.

```
Algorithm ALTERNATE(DIALOG_NAME     :string      ;COMMIT_LOCATION:code_address
                   ;FAIL_LOCATION  :code_address;TEST_LOCATION  :code_address
                   ;END_OF_SEQUENCE:code_address                             ) ≡
    if self.dialog_cleaned_up then
        self.dialog_cleaned_up := FALSE
    else
        <<<CANCEL_TIMERS>>>
        loop
            LEAF_SET := {d ∈ SUBJECT_SET |
                                   {c ∈ d.children | c.active} = ∅ }
            exit when LEAF_SET = ∅
            ∀ D ∈ LEAF_SET loop
                <<<EXTRICATE_PARTICIPANTS>>>
            end loop
        end loop
    end if
    <<<ENTER_DIALOG>>>
end Algorithm ALTERNATE
```

Algorithm ALTERNATE

Figure 4.15

The loops of lines 8 through 15 were not needed in algorithm COMMIT because there could be no remaining nested dialogs. The processes are eased out of the nested dialogs largely so the recovery points can be used in restoration and can be discarded where appropriate in reverse order of their establishment. The SUBJECT_SET was derived in the expansion of CANCEL_TIMERS, above. On each iteration of the outer loop, the LEAF_SET contains the leaves of the tree of `active dialog_entry`s rooted in the executing process' current dialog.

4.3.6. Algorithm LAST_DITCH

The only difference between algorithms LAST_DITCH (Figure 4.16) and ALTERNATE is that in algorithm LAST_DITCH the executing process does not enter a new dialog.

4.3.7. Algorithm TIMEOUT

Algorithm TIMEOUT (Figure 4.17) is executed by the support system when a timer event expires and is found to be related to the TIMEOUT part of a `dialog_sequence`. The process whose TIMEOUT interval has expired is directed to execute algorithm LAST_DITCH at the TIMEOUT part for the `dialog_sequence` whose TIMEOUT interval has expired, and is made the next process to be scheduled for execution. There, that dialog, and all dialogs nested within it are made to fail, and other timer queue entries associated with these dialogs' `dialog_sequence`s are removed.

It may seem irresponsible to change the process' `current_dialog` so directly (lines 5 and 9), but it does not matter where the process had been executing. If the process had not been executing the outermost dialog in the designated `dialog_sequence`, the dialog it was executing will be properly terminated by algorithm LAST_DITCH. The asynchrony of this algorithm with the processes using the other algorithms is a major reason for the ''non-interruptibility'' of parts of the other algorithms. Some further processing is really needed since multiple timing intervals could expire at the same time, but the

```
Algorithm LAST_DITCH ≡
    if self.dialog_cleaned_up then
        self.dialog_cleaned_up := FALSE
    else
        <<<CANCEL_TIMERS>>>
        loop
            LEAF_SET := {d ∈ SUBJECT_SET |
                                   {c ∈ d.children | c.active} = ∅ }
            exit when LEAF_SET = ∅
            ∀ D ∈ LEAF_SET loop
                <<<EXTRICATE_PARTICIPANTS>>>
            end loop
        end loop
    end if
end Algorithm LAST_DITCH
```

Algorithm LAST_DITCH

Figure 4.16

```
Algorithm TIMEOUT ≡ -- (assumes THE_QUEUE_ENTRY has been removed from  )
                    -- ( timer_queue and THE_QUEUE_ENTRY.kind = TIMEOUT)
   --<make THE_QUEUE_ENTRY.the_process runnable>
   if THE_QUEUE_ENTRY.surrounding_dialog = NULL then
       THE_QUEUE_ENTRY.the_process.current_dialog := d where
                (d ∈ dialogs_table |
                 THE_QUEUE_ENTRY.the_process ∈ d.participants)
   else
       THE_QUEUE_ENTRY.the_process.current_dialog := d where
                (d ∈ THE_QUEUE_ENTRY.surrounding_dialog.children |
                 THE_QUEUE_ENTRY.the_process ∈ d.participants)
   end if
   THE_QUEUE_ENTRY.the_process.next_instruction :=
                THE_QUEUE_ENTRY.timeout_part
   --<schedule THE_QUEUE_ENTRY.the_process next!>
end Algorithm TIMEOUT
```

Algorithm TIMEOUT

Figure 4.17

presentation is being kept simple.

4.3.8. Algorithm IMPLICIT_OTHERS

Algorithm IMPLICIT_OTHERS (Figure 4.18) shows how easy it is for a compiler-generated handler for otherwise unhandled exceptions to find the appropriate exit from the enclosing dialog. This would also be generated for `raise` without parameters and for the FAIL statement.

An exception can be raised by hardware, user software, the FAIL statement, or language support software. The placement of the calls to `push` and `pop` for the `failure_continuation` stack respectively before and after each dialog ensures that an exception raised in either of the DISCUSS statements of the example including the acceptance tests causes failure of the associated dialog, while an exception raised elsewhere in the `dialog_sequence` causes failure in the surrounding context.

4.3.9. Algorithm BRANCH

A process does not have to "fall through" to the end of the DISCUSS statement in order to complete it. A GOTO statement, "exit loop" or "next iteration" construct with a destination external to the dialog sequence, or a RETURN statement are alternate methods of exiting a DISCUSS statement. If the RETURN statement has a result expression to evaluate, that is done while within the dialog.

```
Algorithm IMPLICIT_OTHERS ≡
   goto top(self.failure_continuation)
end Algorithm IMPLICIT_OTHERS
```

Algorithm IMPLICIT_OTHERS

Figure 4.18

The usual exit from a successful dialog is via the success_continuation. The success_continuation as set up upon dialog entrance is not likely to lead to the destination of one of the alternate forms of normal exit from the DISCUSS statement. The possibility of failure of some acceptance test between the source and destination of the branch prevents use of the success_continuation by overwriting its values. Algorithm BRANCH (Figure 4.19) and a few lines in other algorithms provide an alternative that does not erase the success_continuation stack in case the decision to take the branch must be revoked.

Recall that the test_address stack holds the beginning of the use of algorithm ACCEPTANCE_TESTS for each DISCUSS statement corresponding to an active dialog in which the process is participating. Thus it holds the process' itinerary on its way to the branch destination. Until the destination is reached (algorithm COMMIT line 7 and macro EXTRICATE_PARTICIPANTS line 21) or the decision to take the branch is revoked (macro EXTRICATE_PARTICIPANTS line 10), algorithm COMMIT (line 8 for the process terminating the dialog, and in macro EXTRICATE_PARTICIPANTS line 22 for other processes) is to use a

---

```
Algorithm BRANCH(WHERE_TO:code_address;HOW_MANY_LEVELS:natural) ≡
    -- (parameters: must cross HOW_MANY_LEVELS dialog boundaries to WHERE_TO)
    COPY_STACK := self.test_address
    loop HOW_MANY_LEVELS times
        push(TMP_STACK,pop(COPY_STACK))
    end loop
    if ¬ stack_is_empty(TMP_STACK) then
        TMP := pop(TMP_STACK) -- throw top element away
    end if
    push(self.branch_stack,WHERE_TO) -- replace it
    loop HOW_MANY_LEVELS times
        push(self.branch_stack,pop(TMP_STACK))
    end loop
    goto pop(self.branch_stack)
end Algorithm BRANCH
```

Algorithm BRANCH

Figure 4.19

---

stack of alternate success continuations here called the `branch_stack`. The `branch_stack` is constructed to contain the final destination as the bottommost item (line 10) and the `test_address`es for the intervening DISCUSS statements (line 12). Once the stack is constructed, the first jump on the journey is taken (line 14).

The construction proceeds as follows. The stack of `test_address`es is copied (line 3) lest it be destroyed by the call to `pop` on line 5. The loop on line 4 inverts the part of the copied `test_address` stack pertinent to the intervening DISCUSS statements. This inversion allows access to the bottommost address which is changed (lines 7 through 10) since, upon leaving the last intervening dialog, the process is to continue at the final branch destination rather than the acceptance tests of the DISCUSS statement surrounding that dialog's DISCUSS statement. The desired `branch_stack` is built by reversing the inversion process (lines 11 through 13).

4.4. Conclusion

We have outlined the more difficult or unusual parts of an implementation of the `dialog_sequence` thus demonstrating that the dialog and colloquy can be implemented. We have neither addressed efficiency nor given detail sufficient for an ''implementor's guide'' as these are inappropriate to the demonstration.

CHAPTER 5

More Fundamental Problems

The dialog and colloquy concepts, as embodied in the `DISCUSS` statement and `dialog_sequence`, provide implementable answers to our initial objections to other backward error recovery proposals as discussed in Chapter 2. The dialog and colloquy ideas have many good points. As pointed out in Chapter 3, these ideas afford us the error detection flexibility of multiple acceptance tests. Also in Chapter 3, we showed that these ideas 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. In Chapter 4, we showed that these ideas are implementable. A syntactic proposal for merging these ideas into an existing language for the targeted applications area is described in Appendix A. And, Appendix B demonstrates that all previously proposed programming language facilities for backward error recovery are special cases of the use of these ideas.

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 programming language with no further concern.

The merging of dialogs into a real language, however, can reveal semantic difficulties not readily apparent in the general discussion of the idea. Certain aspects of actual programming languages seem to conflict with the goals and design of backward error recovery facilities. In this chapter, we examine the problems which arise in attempting to merge backward error recovery, as represented by the dialog and `dialog_sequence`, into a modern programming language. This examination discloses several new problems with backward error recovery in real languages. The conflicts are problems of a more fundamental nature than those addressed earlier in this dissertation because, although researchers in backward error recovery have failed to recognize them, the backward error recovery concept cannot become useful without addressing them.

It is important to realize that these problems are not specific to the dialog and colloquy. They arise because of the fundamental requirements of backward error recovery in concurrent systems, and would occur with the conversation also. We use the colloquy merely as an example. In general, other researchers have either assumed these problems would not occur, or based their research on programming languages with very limited (and impractical) facilities, such as CSP [Jalo86]. The problems we discuss here do not arise in those languages because of the relatively simple semantics that such languages have. No previous research in software fault tolerance has examined the realities of introducing backward error recovery into a language designed for real applications. The foci of the other approaches have been so narrow as to be *naive*.

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.

The specific programming language we use is Ada. In examining problems with including backward error recovery in Ada, we do not condemn Ada. Rather, we show the immaturity of the backward error recovery approach in relation to languages of which Ada is one example. Often, the words "process" and "task" are used in what may seem an interchangeable manner. We use "task" to indicate the specific incarnation in Ada of the idea of a "process".

Several new definitions will be necessary. A well-known source of difficulty in extending backward error recovery to concurrent programs is that a set of processes must agree about establishing and discarding their individual recovery points. A set of processes with coordinated recovery points, is a *coordinated set*. The formation of a coordinated set is the major goal of both the conversation and the dialog. The part of a process' execution from the establishment of the recovery point until it it discarded is termed a *recovery region*. The recovery points for a coordinated set are referred to as a *recovery line*. The set of participants in a dialog constitutes a coordinated set.

5.1. Smuggling

Recall that 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.

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, the first form of smuggling, that 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 (by our definition) in Ada. The dialog as defined and shown implementable in preceding chapters prevents smuggling via messages for an Ada-like language. A more general message-based language would present more problems for backward error recovery.

Smuggling is usually *assumed* to be controllable. Like others, we have so far ignored the many other means of smuggling. The second form of smuggling, that through implicit information flows, is much more involved. By *implicit* information flows we mean methods by which one process gains information about another process' activities or status without using the explicit communications statements provided in the language. To describe the problem, we separate implicit flows into 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 major 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 second category, unexpected implicit information flows, is represented by process manipulations. Ada allows processes to be manipulated in two principal ways. These are task creation and task destruction. 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. As such, it deserves special consideration apart from other means of implicit information flows.

5.2. 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.

5.3. Summary and Outline

The language facilities described in Chapter 3, Chapter 4, and Appendix A seem on the surface to be adequate for backward error recovery, however they turn out to be incomplete solutions to these problems. At this point, one may consider them to have been presented as a basis for further discussions, although they solved all of our objections to previous proposals.

This and the succeeding three chapters are closely related. In those chapters, we expand upon and address the problems which we have introduced here. Where we have been able to find a solution, we describe it in detail. Where a seeming solution is unacceptable, we explain why. And, where a solution is academically sound but would be rejected in practice, we present both the ideal solution and a compromise.

In the next three chapters, we deal with these problems in roughly the same order as they were illustrated in this chapter. These examinations serve to illustrate and elaborate the ''more fundamental problems'' introduced above. The problems are discussed under the headings *shared data*, *process manipulation*, and *program structure* in subsequent chapters.

Chapter 6 discusses smuggling due to the category of expected mechanisms for implicit information flows, first via single- and then via multiple- path sharing. To do this, Chapter 6 returns to the mechanisms which were used as illustrations in this chapter. They are, shared variables, aliasing, and pointers.

Chapter 7 explores smuggling due to unexpected mechanisms for implicit information flows. It deals first with task creation. A solution for the most general class of tasks is presented, and special cases of the solution are provided for the other classes of tasks. The chapter next deals with task destruction. Normal termination is found not to be a problem. Aborted tasks and their implications for task attributes embody the bulk of the investigation.

Finally, Chapter 8 elaborates on the problem of backward error recovery's conflicts and incompatibilities with the language's communications philosophy and with modern programming precepts. Chapter 8 describes a new compromise between backward recovery and service tasks.

Before proceeding, recall that it does not matter whether smuggling is considered malicious or inadvertent. The central reason for preventing smuggling is the limitation of the extent of corruption by invalidated data should backward recovery be required of the originator of the data.

CHAPTER 6

Problems with Traditional Sharing

Here, we address smuggling through the category of mechanisms for implicit information flows that one would expect to find available in a programming language. This category is represented by sharing of data. Sharing of data occurs through many mechanisms, and we illustrate with three specific mechanisms. They are what one would normally call shared variables, the special problems of aliasing, and pointers.

A principal means of implicit communication between processes in most programming languages is the use of shared objects. A shared object is any changeable, non-process, non-procedural object that is visible or accessible to more than one process or to procedures callable from more than one process. A shared object that has only one access path or name is called a *shared variable*. In sequential programs, these are called global- or non-local- variables. Shared objects can have multiple names or access paths, either intentionally through *pointers*, or unintentionally through *aliasing*. Ada provides single-access-path and both forms of multiple-access-path shared objects.

We could choose to disallow shared objects completely in a language for backward error recovery. That would be unrealistic since one simply *is* going to want exceptions to such a rule. The fact that Ada permits shared objects is an indication that they are still considered vital by the community. We do not contend that sharing is essential, only that it seems to be in such demand that we need to attempt to deal with it rather than simply define it away.

Besides habit, there is the practical reason for sharing — fast access. Communicating via messages with a process that guards the shared information can be too slow. In addition to the delays that can arise through complex message protocols, there is the potential of incurring further delays in copying parameters, particularly large tables and arrays. The Ada LRM *allows* but does not require passing arrays by reference. This permission only benefits the callee (who in Ada would be the guardian of the ''shared'' array, not a user of it), and that only during a rendezvous.

Some of the categories of information that tend to be shared within a control system are:

(1) The most recent, perhaps digested, readings from sensors. This would include both a consistent set of readings and a partial set as a new consistent set is being acquired.

(2) The most recent settings or values sent out to the actuators or other external lines. One reason for retaining this set of values is to enable the temporary continued service algorithm of placing the most recent outputs again.

(3) Tables. Often, portions of control systems operate as table-driven programs. Each table might be interpreted by a single process, parceling out the information to other processes upon request. This normal operation might raise doubts about whether the table is shared. However, systems designed for long-term missions would also provide for other processes to update or alter the table when objectives change.

(4) Digested status of the software system. This includes such constructs as dynamic networks (pointers).

(5) Knowledge of current goals and progress that has been made towards them. The extent of this may depend upon the size of the system.

We explore shared variables, aliasing, and pointers in turn.

6.1. Shared Variables

In this section, we examine the problems involved in trying to merge backward error recovery with the use of shared variables. Global variables are not a problem for the original recovery block and recovery cache work, since those proposals did not address concurrent systems. In concurrent systems, global variables become shared variables with problems of concurrent access and update. With the `dialog_sequence`, we have already limited explicit communications statements to occur only within DISCUSS statements. A desirable goal of a solution is that any updating of shared variables that the language allows should also be so protected.

The DISCUSS statement scheme, as described so far, has several shortcomings in relation to shared variables. We will use its shortcomings to illustrate how insidious the shared variable problem is. In so doing, we also show that this aspect of the DISCUSS statement scheme is sorely inadequate as a solution to data sharing. For example, to enforce its rules, we need to be able to detect all shared variables. In addition, we show that activities within nested dialogs can interfere with each other, and that suspending outside tasks during dialogs can lead to deadlock. Further, the SHARES lists can have intersecting sets of names, and new access paths need to be found for each instance of a dialog. There are also some concerns left over from the nested atomic action literature that need addressing. Finally, we could question the global acceptance test itself. We elaborate on each of these shortcomings below, after which we proceed to seek better solutions.

Detection of Shared Variables

Before anything can be done about shared variables, it must be possible to determine which variables are actually shared. The intention of the dialog declaration, particularly the name list following the keyword SHARES, was to force shared variables to be explicitly declared as such (as **`pragma shared`** was apparently intended to do). The dialog name in the DISCUSS statement thus served as a form of shorthand for the import list. The processes entering the dialog were able to use the variables then and only then. The function forced the designer to provide a uses list and told the compiler explicitly what to cache.

Unfortunately, the rules of Ada as they stand allow sharing of other variables beside those explicitly declared with the dialog function. It is not always easy to determine which other variables are also shared.

Suppose all shared variables could be detected. Enforcing the use of dialogs for references would mean that the task that declared a variable, x, would have to enter a dialog whose SHARES list includes x every time it needed to reference x, just as other tasks would be required to do. Otherwise, while other tasks use x within a dialog, the declaring task might blithely access x as if it were a local variable. Non-protected references need to be detected and flagged as "errors" before execution time. We base our notion of sharing on the ability of multiple tasks to reference a variable. If we want to enforce the use of dialogs around references to shared variables, all possible references must be detected. Detecting all references can entail extensive compile-time analysis. This analysis may be as involved as is aliasing detection.

Suspension and Asynchronous Entry (Interference)

In the DISCUSS statement scheme, we assumed the approach of suspending tasks or dialogs that attempted to compete with an ongoing dialog for use of shared variables. There are several disadvantages of suspending competitors for use of an item in a SHARES list, specifically:

(1) The ongoing dialog's effects can be nullified. Suppose both the dialog and the outsider want to change the object. The outsider is released from suspension and overwrites the object immediately after the updating dialog ends.

(2) There is a limitation on parallelism imposed by the suspension, particularly if the dialog takes a long time.

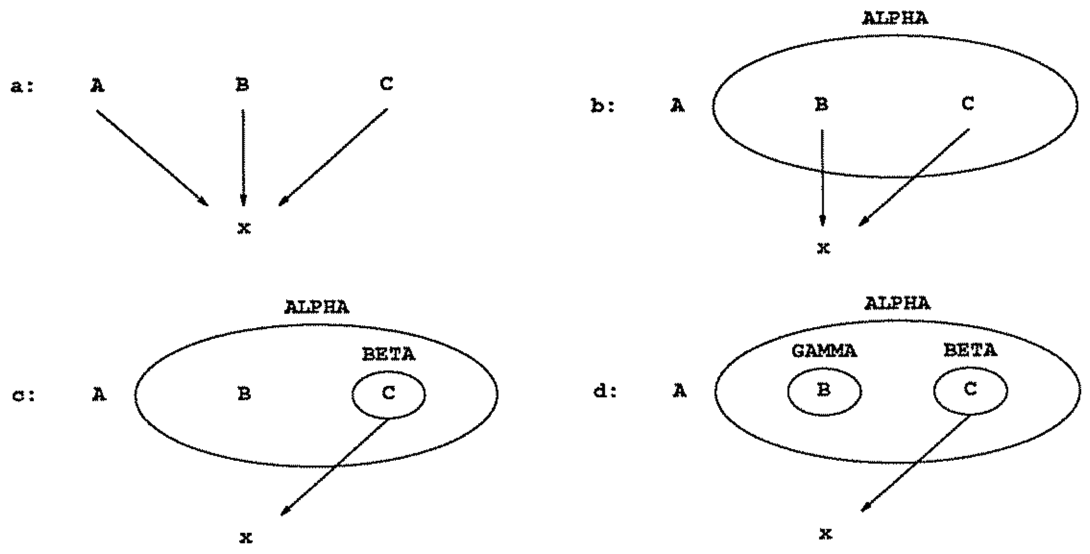
(3) The possibility also exists that other outsiders could notice the first outsider's suspended status/behavior, thereby recognizing that the dialog is in progress— a violation of the strict definition of atomicity for the dialog.

(4) Two dialogs featuring the same objects in their SHARES lists could deadlock. An implementation might allow each to lock a variable needed by the other, thus suspending both dialogs until one or the other times out, if one does at all.

(5) The suspension approach in combination with asynchronous dialog entry also permits deadlock. Suppose the planned last action of an outside task before it enters the dialog is to access the object. The task's suspension could prevent it from ever entering even though its presence may be vital to the success of the dialog.

(6) Finally, the suspension policy employed for DISCUSS statements entails the idea of multiply locking variables. Consider the situation shown in Figure 6.1c. Here, we have tasks A, B and C with A outside any dialog, B and C within some dialog ALPHA, and C within dialog BETA within ALPHA. A variable, X, in the SHARES lists for ALPHA and BETA must be locked so that A cannot use it, so that B and C can use it before and after BETA (Figure 6.1b), and so that C but not B can use it during BETA. A lock cannot be a simple count of nesting level since, as in Figure 6.1d, B may try to enter a different dialog, such as GAMMA, within ALPHA while BETA still exists.

Given all these disadvantages, it is clear that suspension is not a viable solution. If we give priority to the proposition that a dialog is an atomic action, it is inconsistent to block an outsider task or dialog at all.



Permitted Shared Variable Accesses

Figure 6.1

Cache Referencing

Space for a shared variable is not necessarily allocated with the local variables declared along with it. Once a recovery line is established, it may be more difficult or costly than before to access the shared information.

The version of the shared variables referenced by the dialog function changes dynamically. Each time a given dialog function is called, it may have to ''look'' into a different cache or ''place'' within a single cache. Due to references to prior values within the dialog function, it does not matter whether the mechanism for the recovery cache stores updated values to overwrite prior values on commitment or stores prior values to overwrite updated values on recovery. The compiler writer has to change the way the code finds paths to objects by referencing caches instead. This is only a hint of the kinds of difficulties that may be involved.

Nested Atomic Actions

The problems we have been examining with shared variables seem on the surface to resemble the problems with shared resources in the database literature [Haer83, Kohl81, Bern81, Gray81, Bern78, Gray76, Eswa76, Best80], but they are very different. Much of that literature deals with a single participant in any one action protecting its activities from the view of other processes or actions; whereas we are dealing with multiple processes entering the same action asynchronously and locking out non-participants while endeavoring not to interfere with activities of participants before they actually enter. It is not possible to identify an action exclusively with one process as in that work. Some of the database work does consider nested actions within a single action proceeding in parallel under separate processes, but most of those processes would be created and destroyed during the outer action as they would have been created solely to perform the nested actions [Lync83, Gray81, Lisk83]. Still, never were two processes equal partners in an action that was ''shared'' between them.

There is a concern in the database literature that does relate to this discussion. That is the granularity of locking available in a database [Gray76, Kohl81]. We have concentrated on locking whole variables, not fields of records or individual array elements. Still, there may be problems with locking the database of shared variables on a word-by-word basis. This level of granularity may not yet be implementable. Even so, due to nesting of dialogs and the compile-time information in the SHARES lists, it may be possible to arrange a special storage allocation scheme that would group items in a SHARES list together, and to group sub-lists within these groupings. The outermost level will have locked a set of variables that contains the variables shared by any inner level.

The Global Test

As a final problematic area, the global acceptance test is a good idea, but avoidance of shared variables is also a good idea. These two good ideas are in conflict. Without shared objects the global acceptance test would have very little to check. It might be used to check on outside conditions to be sure the whole colloquy was "appropriate", but could not check on its results because all values would be hidden as locals within the participants. In the absence of some form of sharing, the global acceptance test may have been good only as a concept[†]. Apart from the desire for sharing in the community of applications programmers, the global test is the only need we have found for sharing. We are thus tempted by a solution that disallows sharing.

We have discussed many of the problems involved with permitting sharing of variables in the language. The DISCUSS statement scheme has not addressed them fully and thus is not a solution. We next show some alternate schemes that one might consider as replacements. For each of these, we show that it also is not a viable solution. Following those discussions, we present a sound but impractical solution along with a practical solution to the shared variable problem.

6.1.1. Some Other Non-Solutions

The DISCUSS statement scheme of Chapter 3 is not alone in being flawed with respect to shared variables. Other schemes that one might consider have other manifestations of the shared variable and related problems. We present some representatives of these schemes below.

[†]None of our examples within the text or appendices contain a filled-in global acceptance test. This is due to the fact that they are trivial examples.

Shared Variable Monitors

One alternate approach to solving the shared variable problem is to restrict or ritualize access to them. A method of protecting shared variables from simultaneous update in Ada is to encase them within special processes resembling monitors [Cohe85]. Monitor processes are the preferred means of sharing in Ada. We could use monitor processes to contain and ritualize access to what would otherwise be shared variables.

One would expect the new value of a shared variable to be accessed as part of any global acceptance test. Arrangements would have to be made for the dialog function to read it without blocking during the test. A monitor process must obey the colloquy rules as well as any other process, so the dialog function would not be able to see inside it to access the shared variables. The monitor process would be suspended awaiting the outcome of the global test while the function is being evaluated, hence it would not be able to accept inquiring entry calls from the dialog function.

We might alternatively make the function a special kind of process with special syntax and semantics. It could then "rendezvous" with or look inside the stopped/synchronized participants for the values they own that are needed in the test. However, to do so it would need to know names for all of the participants. This result would eliminate some of the dynamism of the dialog. If there were a limited number of required members of the dialog, they could be the ones being called for the shared information. Even this has drawbacks in that the required members could desert the dialog. Further, as we shall see in Chapter 8, the use of monitor processes introduces still other problems.

Distributed Global Test

As a variation of the above attempted solution, we might try to force any monitor process to include an especially marked boolean function for "correctness" of its data base. This function would be called automatically as part of the global acceptance test upon exit of any dialog the monitor process happens to be in. This makes provision for determining local consistency or correctness within the service task.

This approach is unsatisfactory because it could be only duplication of the local test and because it does not make provision for determining consistency between the databases of multiple monitor processes exiting from the same dialog. It also does not provide for the fact that the shared data in a given monitor would be used in different dialogs requiring different tests as program progress is made.

Parameterized Dialog Functions

Recall that in Chapter 3 we required the dialog function to be parameterless. We could attempt to solve the shared variable problem by restricting sharing to the formal parameters of a modified dialog function. Under this plan, there would be no shared variables in the usual sense. Variables in any declarative part would be visible only to the sequence of statements of the unit containing that declarative part. Within a DISCUSS statement, the scope would also include the formal parameters of the dialog function named by that DISCUSS statement. The formal parameters would be visible to every participant as if they were shared variables. Since the formal parameter names would be visible only within the DISCUSS statements of the participants, the information being communicated through them could not be accessed by any non-participant. Upon successful completion of the dialog, each participant would retain the communicated information in the form of the actual parameters it used in correspondence to the formal parameters with the mode OUT.

Parameterizing the function means dealing with the parameters at the DISCUSS statement within each participant. Each participant must deal with all parameters, even though it may only "want" a few as results or have input for a few others. The worst part of using generalized parameters on the global test function is that each participant might place a different value on the IN parameters.

An alternative is to have only OUT parameters on the function. This is the first plan that has the advantage that no shared values need to be cached upon dialog entry. However, the OUT parameters variant on dialog functions has several drawbacks:

(1) The remaining sharing is very limited. The variables shared are available only for the duration of the dialog and do not retain their values from one incarnation of the dialog to the next. These variables do not fit into the categories listed above of information usually shared in control systems. This kind of sharing may be of benefit only for the global test.

(2) All participants must accept result values for all parameters when each may only want some (or none). Within the DISCUSS statement's text, a participant can simply not use the formal parameter names it does not need. However, it must still provide an actual parameter for each of the formal parameters to receive the result value.

(3) The OUT parameters should be required to have default initializations so the test depends upon the participation of no task. Unless there were some default initialization for every formal parameter, the several participants might rush into each others' way to ''initialize'' the formal parameters. Even with a default initialization requirement for the parameters, it is conceivable that some parameters to be shared would need different initial values for different incarnations of the dialog.

(4) Each participant in the dialog must ignore the local names for the variables it used as actual parameters in favor of the formal parameters' names within the DISCUSS statement. Although enforcible, this new visibility rule is not likely to seem natural to the programmer. To use an actual parameter's name within the DISCUSS statement, particularly as the result of an assignment, would not have an effect upon the shared formal parameter, nor would it have permanent (lasting beyond the successful completion of the dialog) effect on the local variable.

Thus, the idea of parameterizing the dialog function also must be rejected as inadequate and unrealistic.

No Nesting

It is tempting to reason that many of the the problems we have pointed out are due to the nesting of dialogs. To solve the problems, therefore, one might consider abandoning nesting. Rather than develop a complete solution attempt around this idea, we show the flaws in this direction of reasoning.

The elimination of the ability to nest dialogs within each other has some advantages. If there were no nesting of dialogs, many of the problems would go away. We would have no interference of SHARES lists or multiple locking, and there would be no indeterminate nesting depth for task creation/destruction†.

On the other hand, we would only have one level of protection, so the acceptance tests would have to be very good indeed. We would be restricted to program designs that, basically, do not use nesting. We would also be increasing our dependence on the acceptance tests. We could no longer appeal, as does Lee [Lee78], to outer tests catching mistakes in inner tests.

Alternate View of Shared Variables

We might seek a solution to the problem by appealing to an alternate view of shared variables.

Consider the view of a shared variable as many local (to each task) copies updated by messages. This view is known to match the semantics of shared variables, and is even employed as a valid implementation. Under this view, an update to a shared variable is similar to the sending of a broadcast

†We show the importance of this in Chapter 7.

message. The message is broadcast[†] to all processes, including those which will only exist in the future. As the sender of a message, we would have the updator of the shared variable be in a dialog at the time. As recipients of the message, we would likewise have all other tasks (including those which will only exist in the future) be in the same dialog. To do otherwise is to allow smuggling.

The asynchronous nature of tasks makes it difficult to establish a recovery line and certainly to pre-package it with a syntactically enforceable structure in the manner that a DISCUSS statement is enforceable for explicit messages. We might be tempted to provide acceptance tests at the sites of the read accesses of the variable with the readers merely backed to a point immediately before the attempted read. There may be many scattered potential read sites per update, also there would not be any reduction in delays. A given reader may take quite a long time to reach its test. This traps all tasks, both those that have and have not read the variable. It also makes no provision for readers who do not yet exist.

Another Close-But-Failing Solution

As our final non-solution, we might try to retain the suspension approach only for writers. The SHARES list would have modes associated with each variable named. These modes would be similar to those found on parameters to procedures. For procedure parameters, the modes are IN, OUT, and IN OUT. For variables in a SHARES list, the modes would be IN, which would always be assumed, and IN OUT, which would always be abbreviated as OUT. There would never be any mode OUT in the procedure parameter sense. We will henceforth use the abbreviation when speaking of the mode IN OUT.

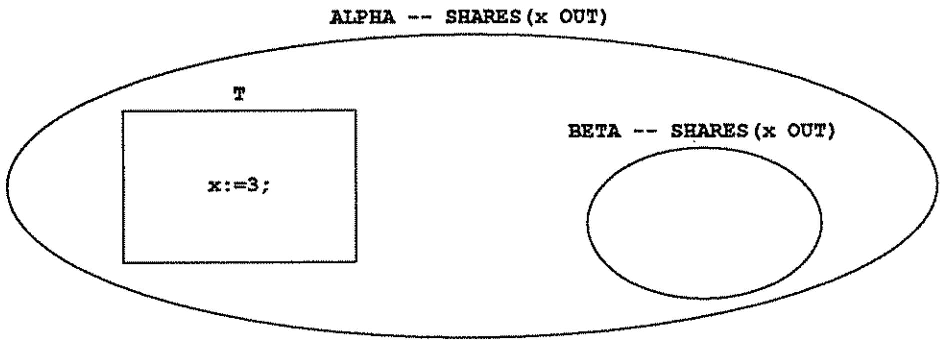
The mode IN would be a statically verifiable assurance that the variable would be subject only to read operations within the designated dialog. That is, the variable would not be the target of any form of assignment. The mode OUT would be an announcement that, on some execution path, the variable could well be the target of some form of assignment within the dialog.

[†]This is a generalization. Strictly, the message is sent to all processes to which the variable is visible or that can access the variable through some path.

For nested dialogs, it follows that all dialogs nested within a dialog using mode IN on a given variable must also use mode IN on that variable. Dialogs nested within a dialog using mode OUT on a given variable may use either mode on that variable. There would be *no* references to shared variables outside dialogs. A writing dialog would maintain a copy of the variable from the first encountered assignment until successful completion of the dialog. After a writing dialog had successfully evaluated its dialog function, it would await completion of all the current reading dialogs before using its copy to overwrite the actual variable. Once a writing dialog reached the wait state just described, no new reading dialogs would be allowed to start until the overwriting copy operation was complete. No two writing dialogs would be allowed to proceed in parallel; a second writing dialog could not start until the first writing dialog had completed its overwriting copy operation. There might be multiple variables in a SHARES list, some with mode IN and some with mode OUT. Recall from Chapter 3 that a dialog starts as the first participant attempts to enter it. Under this plan, a dialog could not start until the IN and OUT conditions above had been met for all of its shared variables.

Under this scheme, it would be possible for a dialog to be timed out as it waits for the readers to quiesce and after it has successfully performed its computation and checked it against the local and global acceptance tests. If we had writer dialogs wait at their beginning rather than at their end, all of a timed-out dialog's work would not be wasted. That variation would severely limit parallelism since we could not then justify allowing any reader dialog to proceed in parallel with the writer.

This kind of scheme may deal with interference between reading and writing dialogs, but it does not deal with interference between dialogs and tasks. We illustrate this with Figure 6.2. Consider a dialog, ALPHA, that names a variable, X, as shared for writing. Nested within ALPHA is a dialog, BETA, that also names X as shared for writing. Finally, one of the participants in ALPHA, but not of BETA, is a task, T, that executes an assignment to X. ALPHA has blocked any use of X other than by its own participants. The participants of BETA are entitled to use X for writing without interference by participants of any other dialog within ALPHA. We have said nothing about how to prevent T's write into X from interfering with BETA. One way to prevent this kind of interference would be to force each task



Task Interference With Dialog Write Locks

Figure 6.2

(such as T) to finally enter a private dialog for each statement in which it uses a shared variable. Besides having an overhead (which we expect to be overpowering), such a "fix" would amount to having the language completely solve the readers and writers problem for the user. This is not what we set out to do. Our purpose is to prevent smuggling via shared variables without allowing the smuggling prevention scheme to itself introduce great data inconsistencies.

There are also severe granularity problems with this scheme. This variation of the readers and writers problem permits multiple writers. In implementing it, we might employ a variation of the database concepts of read locks and write locks. As we shall see, each instance of a variable declared as shared would have to have associated with it a *stack of locks*.

For each instance of the variable, the basic lock itself must tell whether the variable is locked for read or write or is not currently locked. It must also identify which dialog owns the currently locked rights. This identification cannot be a simple pointer to that dialog's `dialog_entry` since we permit multiple parallel reading dialogs. There must also be some method of queuing dialogs that want to lock a variable while a writing dialog proceeds.

Recall the 6th listed disadvantage of suspending competitors under the heading "Suspension and Asynchronous Entry (Interference)" in section 6.1 (page 104), "Shared Variables", "A variable in the SHARES lists for ALPHA and BETA must be locked so that [task] A [outside of ALPHA] cannot use it, so that [tasks] B [within ALPHA but not within BETA] and C [within BETA] can use it before and after BETA, and so that C but not B can use it during BETA." From this, we conclude that there must be multiple coordinated levels of the basic lock. That dialogs are strictly nested suggests a stack of basic locks.

Due to recursion in the language, multiple instances of the shared variable can appear at execution time. Each of these needs its own stack of basic locks. Thus, the variable's lock stack must be allocated along with the variable. It cannot be kept off to the side in some statically allocated memory.

We have seen a need for a stack of very complicated locking data for each instance of a shared variable. This fact suggests that we severely limit the number and size of variables that we permit to be shared, or that we find another kind of solution entirely.

Along with the DISCUSS statement scheme of Chapter 3, up to this point in this chapter we have shown several schemes for including backward error recovery in a language with shared variables. None has fully addressed the problems that shared variables impose. In the next two subsections, we present two solutions, one correct and one practical. Unfortunately, they do not coincide.

6.1.2. An Academic Solution

The problems with the combination of backward error recovery and a realistic programming language speak against the inclusion of any form of sharing in a language meant to include reasonably secure backward error recovery facilities. This is an academic view, that shared variables are incompatible with backward error recovery. The academic solution is to remove shared variables completely from the language, thus removing the problem. Interestingly, the academic solution is also advocated as a solution to different sets of problems encountered by other sub-disciplines, such as formal verification, when dealing with shared variables [Ambe77]. These sub-disciplines have their own reasons for rejecting shared variables that we will not discuss here. We mention them as evidence that shared variables do not cause problems for backward error recovery alone.

Removing shared variables is unacceptable to practitioners designing and implementing real-time systems. They would claim that such a restriction would make the language unrealistic. From the practical viewpoint, people *want* (they say *need*) shared variables. As was noted, there are many kinds of information that are habitually shared in actual control systems. We suspect that, although an argument could be made for each of them that they need not be shared, in practice developers of these systems will insist on the familiar methods.

Thus, a practical compromise solution is required. As a compromise, the solution we present is not ''air-tight''. We cannot enforce disciplined use of the proposed solution, so we cannot guarantee that programs will be free from smuggling. However, when used with some care, this alternative can provide a high degree of tolerance to faults.

6.1.3. IMAGEs

As was shown in Chapter 3 and Chapter 4, a marginal amount of checking can be achieved with the approach of Chapter 3. The practical solution to the shared variable problem, which we call IMAGEs, is a modification of the approach of Chapter 3.

The approach is defined by the following rules.

(1) The scope of a variable must be changed explicitly by declaring it as shared, otherwise the name is not visible to other tasks.

(2) Even by its declarer, a shared variable may only be accessed during a dialog whose SHARES list mentions that variable.

(3) Tasks are not blocked when attempting concurrent access of shared variables. Instead, each dialog has its own copy of the variables in its SHARES list.

(4) Some categories of shared objects will have exceptions to the enforcement rules 1 and 2.

Expanding on Rule 1, all variables are local to the declaring task including, through the call stack, formal parameters and variables declared in procedure activations. This holds for each variable unless explicitly declared as shared. Figure 6.3 shows several examples of statically checkable legal and illegal references to a non-shared variable.

As a result of Rule 2, that is all explicit references must be within a dialog, the programmer cannot create many forms of aliases without encasing them within the recovery envelope. This does not prevent aliasing, but it does make errors due to its misuse recoverable.

```
procedure A is
   X : integer;
   procedure B is
      -- use of X                        -- legal
      end B;
   task T;
   task body T is
      B; -- transitive use of X          -- illegal
      end T;
   begin -- A
      B; -- transitive use of X          -- legal
   end A;
```

Legal and Illegal References

Figure 6.3

Rule 3 substitutes a copying approach for the previous suspension approach to multiple access attempts from within and without a dialog. A given copy is available to any task immediately within the corresponding dialog. The copy is (conceptually) made from the immediately surrounding dialog context as the first task enters the dialog. Upon successful completion of the dialog, any changed variables in the SHARES list are copied back into the surrounding context. These copy operations are indivisible as far as user-level tasks are concerned (they are performed by the support system). A dialog does not provide any other Operating Systems style locks for exclusive access to objects. Any other synchronization for shared variable access is the programmer's problem, as before. Thus, in a sense, the group of tasks in a dialog seem to outsiders to behave as a single task. A possible locus for the copy is in the `dialog_entry` for the corresponding dialog as described in Chapter 4.

The possibility exists that the restrictions of the first several rules are too strict. A possible change in the rules, reserved for future investigation, is only to require writes (changes to shared variables) to be within dialogs.

Rule 4 exists for reasons that will become apparent in later sections and chapters. As an example, such an exception will be made for task names. Tasks are shared objects. We have already required references to tasks for communication statements to occur only within dialogs but do not require them to be listed in the SHARES list. We have yet to discuss references such as those required to create and destroy tasks. Pointers to tasks may be similarly excepted.

Having given and elaborated upon the rules that transform the proposal of Chapter 3 into the IMAGEs solution, we present some justification for calling this choice a practical solution. Before the justification arguments, we remind ourselves of some of the principles of backward error recovery as an approach to fault tolerance.

Principles

The practical application of the backward error recovery approach to fault tolerance lies not in strict adherence to its stated principles, but in the underlying reasoning behind them. A few of these principles that are pertinent are listed below along with statements of the goals that they were formulated to help enforce.

(1) It is not the purpose of fault-tolerance facilities to prevent errors, only to contain and recover from them.

(2) Smuggling out of a dialog must be prevented. The reason for preventing smuggling out of a dialog is prevention of invalidation of information by backup.

(3) One purpose of having many acceptance tests on process interactions is so that correct processes can keep faulty processes in check.

(4) Smuggling into a dialog must be prevented. The reason for preventing smuggling into a dialog is that one wants an unchanging base from which to derive new results. This overrides the fact that information smuggled in is not the information that will be invalidated by backup.

(5) If an error is detected, a previous state that is presumed correct is restored. The point of backward recovery is to erase the effects of the fault. Restoration of a previous state is desirable but not required. It is enough to achieve, i.e. construct, a valid state that might have existed. The method of construction is largely what distinguishes backward and forward recovery.

Keeping these principles in mind, we proceed with their application specific to IMAGEs.

Rationale For IMAGEs

As justification for the decisions taken in devising the IMAGEs plan, we show several approaches one might take to questioning the consistency of the shared data under IMAGEs, and show the reasoning behind how IMAGEs addresses them. Then we address some practicality issues in the same manner. In what follows, when we refer to an ongoing dialog, an "outsider" can be another dialog or a task that is also a member of a dialog within which the ongoing dialog is nested.

Upon successful completion of an altering dialog, the overwriting copy operation takes place under control of the language's support which has the ability to lock out unequivocally any competitors for access to the affected space. Thus, the copy takes place indivisibly, maintaining consistency. If others only use the object within dialogs, they take a consistent copy in with them and it is unaffected by a change in the "real" copy if the change occurs while their dialog proceeds.

The implementation needs to make sure that if a dialog does not change a variable, x, then x does not overwrite the outside copy which may have changed. As Figure 6.4 shows, this has complex implications for consistency. Figure 6.4 represents three scenarios giving the value of x before and after dialog ALPHA. In the first, ALPHA writes x. In the second, ALPHA only reads x while an outside task T writes x. In the third, both ALPHA and the outside task write x.

```
  X = 1
... |ALPHA(start)
... |ALPHA(x := 2)
... |ALPHA(end)
  X = 2
```

```
          X = 1
     ... |ALPHA(start)
T(x := 3)|   ...
     ... |ALPHA(end)
          X = 3
```

```
          X = 1
     ... |ALPHA(start)
T(x := 3)|ALPHA(x := 2)
     ... |ALPHA(end)
          X = 2
```

Desired Consistency

Figure 6.4

As another example, let x be changed outside a dialog, ALPHA, and let a variable, Y, change inside. The acceptance test may check for some consistent relationship between x and Y, but the new values immediately after ALPHA may *not* remain consistent with each other. One might also wonder how this scheme is to deal with a case in which a member of another dialog makes (unauthorized) changes to a variable when that other dialog was formed for reading purposes only. In both of these situations, if an inconsistency arises, it should be caught by an acceptance test on a surrounding dialog, the reader dialog in the latter case. If there is no surrounding dialog, we are no worse off than with shared variables without attempts at backward error recovery. As we stated before, there are limits to the offered protection.

We now look at some of the behaviors of the copy approach. If, during a dialog that has its own copy of a variable, x, changes occur in x on the outside of that dialog, how can we restore a previous state with respect to x for the members of the dialog without also backing up the outsiders? Following Principle 5, we do not. The point of backward recovery is not to restore a previous state so much as to undo the effects of the dialog. Thus, we throw away the dialog's copy of x. The resulting situation from the point of view of the former members of the dialog is as if they had been swapped out† while the former outsiders continued to manipulate x. Rather than restoring an actual previous state, we have achieved a state that *might* have existed.

Now suppose a task witnesses a change in a shared variable, x, before asynchronously entering an ongoing dialog that also names x. The inside copy of x is not overwritten as the task enters. To do so would be to allow smuggling of information into the dialog, violating Principle 4. Allowing x to change from the task's perspective as it enters the dialog is consistent with the idea of the atomicity of a dialog.

Continuing with the example of the previous paragraph, an apparent anomaly might occur if the outside copy of x had changed since creation of the dialog but the inside copy had not. Upon entering the dialog the outside task would seem to see a reversal in time or a reassignment of an older value of x

†We ignore Ada task priorities here.

by some other participant of the dialog. This time reversal anomaly is interesting, but we do not pursue it here. However, it could be food for further investigations.

The concerns of the preceding three paragraphs would be moot if the dialog entailed synchronized entry. Asynchronous entry by tasks allows information to come in with the internal state of tasks about activities external to the dialog. This might strictly be called smuggling. Unfortunately, synchronized entry implies severe restrictions on the behavior of tasks; they have to agree about entry as well as exit. The goal of practicality (or at least usefulness) overrides here. This is the first of our several practicality issues.

During a dialog, we have two ways of dealing with outsiders that attempt to access objects in the dialog's SHARES list. We can suspend the outsider or give the outsider a separate copy of the object. The disadvantages, discussed earlier, of suspending competitors for use of shared variables lead us to choosing the copy approach. On the surface, this choice seems to fly in the face of the practitioners' very reasons for using shared objects, fast access through avoidance of copying. One then asks why it is reasonable to let tasks on the inside and outside of a dialog concurrently "see" different values for a shared variable. For the applications we have listed for shared variables, the values are rather long-lived complicated structures. Changing these structures may take a long time, need checking, and need to seem to occur all at once. Meanwhile, others need access to the old values. The update period is precisely where the practitioner would give in to having a copy, so that is a perfect place for a dialog. Notice that an actual copy need not be made until the object is actually assigned either by those within the dialog or outside it.

Since we call IMAGEs a practical solution, let us consider Ada's representation specifications as an example of facilities that allow programmers to bypass compiler security. One may wonder whether, if a shared variable's address is fixed under representation specifications, the variable does not have to be locked rather than copied. The answer is, no. One wants copies of the value at the specified location until the program cannot back up. Only then has a final decision been made about a new value for the fixed location, probably a control line. On the input side, suppose the variable is at a different-value-on-each-read location. Keeping a copy from a single read as part of a consistent base for the dialog's computation is not unnatural. Since a compiler must already ''know'' which locations are of that ilk, it would need to check against invalid uses such as assignments to them anyway.

The broadcast message equivalence interpretation of shared variables and SHARES lists may seem incompatible. As a broadcast message, we want all recipients to have a chance to veto a set of changes as inconsistent. Unrestrained sharing allows only the writer or broadcaster a veto. The SHARES list scheme is a compromise. A group of tasks, some writers and some readers, build a wall around themselves by entering a dialog. Messages are coalesced at the wall, checked for consistency, then sent. Outgoing messages are delayed until the tests by the updator(s), while simultaneous use by outsiders on their own copies remains available. Recall Principles 2 and 3. The central reason for preventing smuggling is the limitation of the extent of corruption by invalidated data should backward recovery be required in the originator of the data. One purpose of having many acceptance tests on process interactions is so correct processes can have a check on faulty processes. As a compromise, we are not requiring *all* processes to have that opportunity.

We have addressed several consistency and practicality issues as justification for the decisions taken in devising the IMAGEs plan. Besides these, IMAGEs also resolves each of the faces of the shared variable problem that we discussed in section 6.1 of this chapter.

6.1.4. Resolving The Earlier Problems

We show in turn below how IMAGEs addresses each of the problems discussed in section 6.1 proper.

Detection:

To be perfect, an implementation of IMAGEs must detect statically all references to variables and protest both references by non-declaring tasks to variables that are not declared as sharable and references by any task outside of DISCUSS statements to variables that are declared as shared. The extension to procedures' use of formal parameters and non-locals must be by pessimistic, i.e. guilty unless provably innocent, restriction based on all points of call and all actual parameters. IMAGEs allows direct references to shared variables only within DISCUSS statements and explicitly designated functions. All indirect references must of necessity be through (or derivable from) these direct references.

Suspension vs. Copy:

We have already argued the merits of copy over suspension for competitive uses of shared variables. Several forms of deadlock were mentioned in section 6.1 proper under the heading "Suspension and Asynchronous Entry (Interference)". Under the copy scheme of IMAGEs, all of them become impossible. The fact that a dialog is in progress becomes transparent to outsiders. And the absence of blocking does not restrict potential parallelism. Finally, there is no need for multiple locking of shared variables since IMAGEs uses a copy rather than a suspension approach.

Cache Referencing:

IMAGEs avoids the caching problems. For explicit source references outside of DISCUSS statements, the compiler can employ the usual fetch and store paths. For other references, the current appropriate storage location for a variable is within the dialog structure designated by the task at any given time.

Nested Atomic Actions:

The copy approach of IMAGEs avoids the granularity problems of databases. During a dialog, the pertinent copy of each variable in the SHARES list resides in the dialog structure for that dialog, so locking of these variables is not needed. At the beginning and (successful) end of a dialog, the copying is under the control of the support system which is capable of suspending the scheduling of processes until it is finished.

Global Test:

IMAGEs preserves the ability to use global acceptance tests on dialogs. We do not eliminate sharing but enforce restrictions on it while allowing the dialog function access to the shared objects.

In summary, IMAGEs is a practical compromise between the demand for shared variables and the need to prevent smuggling when including a backward error recovery facility within a programming language. Having dealt with the problem of smuggling through expected implicit information flows via shared objects with single access paths or names, we proceed to those with multiple access paths.

6.2. Aliasing

Aliasing is one of the two kinds of expected implicit information flows through shared objects with multiple access paths.

Current technology cannot detect all aliasing while providing the range of features demanded by the community. Thus, Ada for example attempts to rule out aliasing by defining programs that use it as ‘‘erroneous’’. To avoid the overhead costs of copying parameters, Ada allows call by reference in some cases for IN OUT mode. Other parameters use copy/result in an attempt to eliminate aliasing. However, even in Ada, aliasing still exists. We do not pretend to have found a way to prevent aliasing where the combination of the community has not. We also find it disconcerting merely to define any program that uses aliasing, whether detected or not, as ‘‘erroneous’’.

A side effect of the IMAGEs solution to shared variables is the elimination of the use of most forms of aliasing as means of smuggling, although aliasing per se is not prevented. Hence, IMAGEs has fortuitously made the problem of smuggling through aliasing moot. IMAGEs has not done the same for the problem of smuggling through pointers.

6.3. Pointers

Another implicit communication problem lies in dynamically allocated objects other than processes. This is another form of communication through shared data where the objects can be accessed via different paths. The multiple access paths phenomenon presents problems in addition to those dealt with under section 6.1, ''Shared Variables''.

As Chapter 7 will show, dynamically allocated *processes* may be brought under the protection and control of backward error recovery. It is not so easy for *general* dynamically allocated objects. Part of the difference lies in the structures created using these objects. In Ada, for example, we know the structure of the task dependency graph and can plan around it, but the general structures that a programmer will compose of access objects are not likewise knowable. Also, the modes of access to these objects are different. For Ada tasks, only ritualized operations are available, whereas the programmer can provide whatever operations he chooses for the general access object (including simultaneous access). Further, processes tend to communicate *with* dynamically allocated processes, and *through* other dynamically allocated objects.

At first glance, the pointer problem looks like the shared variable problem, but the entire dynamic structure needs protecting in order to protect any object within it since any access variable of the appropriate type can be used by any process to access the object. The shared object is potentially accessible via many names. Worse, the collection of names accessing the object is not knowable statically, since which objects are reachable via those names may change, and since many of the names will be fields of other dynamically allocated objects. Unlike simple shared variables, we cannot effectively "lock" the whole data structure (network) since any process may have retained a pointer into it.

6.3.1. Approaches to Solutions to Pointers

A great deal of work has been done in the areas of avoiding faults and detecting errors involving pointers [Lome85]. Although not created to facilitate backward recovery from errors, some of the mechanisms due to that work are suggestive of approaches to the problem of smuggling through pointers. Those familiar with that work may recognize forms of "tombstones" and of "scoped pointers" in the approaches described below.

As with the previous subjects, there are several approaches one might try to deal with smuggling via pointers. We list an undesirable choice, an inapplicable appeal to another area of fault tolerance, an unfortunately unworkable approach, and a direction that seems to us promising.

Ignore Them

The solution adopted by other academic sub-disciplines is to remove pointers from the language. This is also the solution of one camp of practitioners. These groups use entirely different reasoning. The academic solution is based on the impossibility of verifying programs that use pointers. The practitioners' solution is based on concerns for running out of heap space and fear of dropping pointers or creating dangling references. Both ignore the utility of the facility.

A second camp of practitioners simply use pointers blindly, relying on presumed correctness of their programs.

We believe there is sufficient utility in pointers to include them in the language. We also cannot prove correct use of them. Our attention is directed toward limiting the extent of errors in which they might be involved so that programs can recover from those errors.

Fault-Tolerant Data Structures

There has been some work in the area of designing fault-tolerant data structures [Tayl80a, Tayl80b]. That work does not include any proposed basis for syntactic enforcement. It is also more concerned with mistakes in manipulations of links within data structures than with access by multiple processes. It certainly does not deal with preventing smuggling information into or out of a backward recovery envelope such as we have in the dialog[†]. Indeed, the proposed recovery is forward.

Encapsulation

The proponents of Ada urge that a single task, or a package guarded by a single task, have sole access to complicated structures of access objects. The service task then has to obey the dialog rules to communicate with its users, thus obtaining the protection of backward error recovery. Only if the network is guarded by a monitor process, can we grapple with it. This, in effect, would make the dynamic structure "local" to that task. There is no guarantee that programs will employ such restricted usage.

Under this non-guaranteed solution, each pointer type is encased with its object type in a task. The protecting task exports iterator [Lisk77] entries and passes out only virtual pointers or symbolic links to outside users of the allocated objects. If the protecting task fails, all that it protected would be lost. This solution is at least cleaner than allowing random tasks to damage parts of the network in the middle.

[†]From [Blac81] p.151, "In contrast to recovery blocks, the in-line checks contemplated here are concerned primarily with the correctness of the data structures passed to a routine, not with checking the correct functioning of the routine itself."

What exists of the network is all or nothing.

Multiple Heaps

One proposed implementation of allocated objects leads us to consider another approach to solution. It has been suggested that there be a separate heap for each allocatable type. Under this regimen, it *would* be possible to lock the entire heap, blocking non-participating processes' use of any allocated object of that type. One of the undemonstrated "extensions" [Shri79] to the recovery cache for concurrent systems was that of designating "virtual address ranges" for each process for the hardware cache to know about. Although the granularity of locking may be considered too coarse.

Suppose the name of the access type were included in the SHARES list for dialogs. This calls for an exception to the copying rule used earlier for IMAGEs of single-access-path shared variables. The problem with a copy here is the fact that the allocated objects, not the pointers themselves (many of which are local to the individual processes), would be copied. Backward recovery (or even commitment of a dialog) would allow situations to develop in which an outsider's pointer that used to point at an allocated object suddenly ceases to do so as the object's image is overwritten. We do not want the measures intended for backward error recovery to instigate errors.

The alternative of finding all of the pointer variables is the same problem as we started with. One advantage of locking the entire heap would be the opportunity for employing the results of the fault-tolerant data structures research in constructing acceptance tests. Their audit codes could be executed as part of the commitment determination. A concern that would bear looking into is whether this would entail locking all heaps involved in networks consisting of more than one record type.

We do not carry investigation along this line any further, but we do feel this is an appropriate avenue for future work. Carrying the allocated object *type* into a dialog is really no more restrictive of concurrency than having an encapsulating process entering a dialog. The motivation for concentrating on the allocated objects is that it is very difficult to find and protect the pointer-valued objects as a group. For single-path shared objects, the name (access path) and the targeted object are effectively one. For multiple path shared objects there is an intermediate value which itself can be changed. This is the value of the pointer.

6.4. Summary

We have described the problem of smuggling via expected implicit information flows provided by existing, useful language facilities. We have elaborated on this problem both through exposition and by presenting naive approaches to solution along with their shortcomings. We have presented a workable compromise solution for single-access-path shared objects and for unintended creation of multiple paths to objects. We have also given our beginning thoughts on a possible future solution to intentionally multiple-access-path shared objects.

All of our proposed solutions are stop-gap measures for a particular class of programming languages exemplified by Ada. The proposed solutions are inelegant and unlikely to handle even slight variations in language design. To illustrate this, consider the PL/I ''based'' variable.

Based variables entail aliasing through dynamically changeable pointers into non-dynamically allocated space. Our approaches to pointers would not touch based variables' smuggling problems since based variables do not necessarily depend on dynamic allocation. Neither would our solution to shared variables help, since it assumes a single, static access path for each. We bring this up only to show what specific, fragile contrivances these solutions are.

All of this shows that combining backward error recovery with these language facilities produces serious problems not previously addressed or even acknowledged in the literature. It is becoming clear that backward error recovery cannot be included in a programming language after the fact. It must be a design consideration from the outset.

CHAPTER 7

Process Manipulation Problems

We turn now to smuggling through the category of unexpected mechanisms for implicit information flows. We use process manipulation to illustrate this category and, once again we use Ada as an example programming language. Ada allows processes to be manipulated in two principal ways; task creation and task destruction. Ada also provides a means by which a process' execution state can be examined. We examine each of these in turn.

7.1. Smuggling Through Unexpected Implicit Information Flows

One does not normally expect process creation and destruction to be involved when looking for methods of information flow in a language. However, knowledge that a process is created or destroyed constitutes information about the existence and progress of that process *and* information about the progress of the process responsible for its creation. If a newly created process has to be ''uncreated'' as part of backward error recovery in its parent, any existence and progress information that might have been communicated to other processes is invalidated. This could have grave consequences if, as would be expected, they based their further actions upon this information.

Dynamic process creation is an important programming facility. It is also a form of *implicit* communication and often involves explicit communication. Further, it is essential to be able to recover beyond the decision to create a process.

The relative independence desired for processes conflicts with the strict nesting structure we have created for dialogs. As one example, suppose the initial communication between a new process and its creator is couched in terms of DISCUSS statements *as described so far*. The new process is then also entrapped in the creator's set of nested dialogs, however deep. This might make the `dialog_sequence` structure a hindrance in certain applications.

Dynamic process destruction is also useful, particularly in light of the desire to replace faulty processes. Backing up that operation, should the decision to destroy be found in error, presents similar problems.

The following sections deal first with process creation, then with process destruction and its consequences.

Before proceeding, recall that the fourth rule given for IMAGES in Chapter 6 reserves exceptions to the SHARES list rules for some shared objects. Task access variables follow the same rules as task names except when changing values. Access variables to tasks are included in SHARES lists. This is not a requirement for the naming of participant tasks. Many manipulations (such as assignment) of these values are not manipulations of tasks or even communication with co-participants.

7.2. Process Creation

In Ada, one task can create another by two methods. In the first, it elaborates a declaration that specifies the other task. In the second, it executes an allocator where the access object designates the new task. We shall refer to the two tasks involved as **creator** and **created**. The creator-created relationship, particularly in the case of allocators, is *not identical* to the master-dependent relationship explicitly mentioned in the Ada Language Reference Manual (LRM). The reader is assumed to be aware of the similarities and important differences. Using these two methods, tasks can be created in several ways.

(1) A task can be declared directly (Figure 7.1 part 1), or a task type can be declared.

```
(1) task x is entry entry_name... end x;
    task body x is ... end x;
    -- used as x or x.entry_name

(2) task type a is entry entry_name... end a;
    x:a;
    task body a is ... end a;
    -- used as x or x.entry_name

(3) task type a is entry entry_name... end a;
    type b is access a;
    x:b:= new a;
    task body a is ... end a;
    -- used as x.all or x.entry_name

(4) task type a is entry entry_name... end a;
    type b is access a;
    x:b;
    task body a is ... end a;
    ... x := new a;
    -- used as x.all or x.entry_name
```

Equivalence of Declared & Allocated Tasks

Figure 7.1

(2) If a task type is declared, an object of that type can be declared (Figure 7.1 part 2), or an access (pointer) type to objects of the task type can be declared.

(3) Once there is an access type to objects of the task type, an object of the access type can be declared and an object of the task type immediately allocated for the access object to designate (Figure 7.1 part 3).

(4) Alternatively, the allocation can occur within a statement sequence (Figure 7.1 part 4) rather than declarations.

According to the Ada LRM, for most purposes any task created in one of the first three ways could also have been created in the fourth. Thus, the fourth way is the general case.

Task creation is a form of implicit communication since there must naturally be much information transferred between **creator** and **created** (or the support system on behalf of **created**) in order to give **created** sufficient context to execute independently.

The more obvious interactions of tasks during task creation are not really problematic for backward error recovery. There are facilities already built into Ada for detecting and "recovering" from errors during the implied communication involved in creating a task. That kind of communication is also not under the control of the programmer, and is thus not as exposed to designer- or programmer- introduced faults. Many creations take place through elaboration of declarations, making them unreachable by the programmer's code either to create faults or to use a mechanism that we might define. For these reasons, we do not embed that operation in some further fault-tolerant mechanism. Implicit communication during execution of an allocator can be covered by the language-provided exception handling mechanisms. As far as we have embedded exceptions into the `dialog_sequence`, we offer protections for task creations via allocators. However, there is no requirement that the programmer place all allocators within `dialog_sequences` or within blocks protected by bare exception handlers.

The less obvious interactions of tasks shortly *after* task creation *are* problematic for backward error recovery. Information transferred via the implicit communication between **creator** and **created** at the time of creation can be smuggled. This can occur only if one of them later communicates it to a third party who will not be backed up should the creation decision be undone by backward error recovery.

The important interactions of task creation and backward error recovery as described by dialogs become clear when viewed in terms of the states of the creating and created tasks at the time of creation.

(1) **Creator** might be outside of any dialog. The creation operation is then not recoverable. The situation is as it always was in Ada; **creator** cannot be backed up beyond that creation. The decision to create the new task cannot be reversed, and any exceptional conditions that may arise must be handled in a forward, ad hoc manner as **creator**'s (then current) sequence of statements is

abandoned. So the task creation operation cannot be undone, and we need not be concerned with undoing any of **created**'s intervening actions that might otherwise constitute smuggling. It is important to consider a solution that eliminates this possibility. One does not want language facilities that are outside recovery regions.

(2) **Creator** might alternatively be a participant in some dialog. In this case the creation decision is recoverable, but the method of creation and depth of dialog nesting of the creating task manifest other interactions. If for any reason **creator** must (and can) be backed up beyond the task creation operation, **created** must be *destroyed*. It must appear then that **created** *never existed*, and so it cannot have been allowed to smuggle an announcement of its existence outside of **creator**'s dialog. This means **created** can only have been allowed to communicate with tasks that will be backed up with **creator**. We examine below certain important implications this has for the failure of the dialog and for dialog nesting.

Much of the remainder of the task creation discussion in this chapter examines the case in which **creator** participates in a dialog during the creation operation. We first examine tasks created within a declarative part. We next consider more complicated interactions involving tasks created through allocators. Finally, we consider a solution to smuggling due to dynamic process creation.

In all of this discussion, we assume that failure of a dialog entails dissolution of dependents created within it. Thus the dialog does not ''wait'' for them to terminate as would a procedure or declare block.

7.2.1. Declared Tasks

Should **creator**'s dialog fail, all actions due to **creator** must be undone. In particular, **created** must be ''un-created''. Since task creation is a form of communication, we must also consider which dialog, if any, **created** enters as it is created.

Ada's dependency rules for tasks created within declarations, along with the syntactic nesting structure of the language, require that they will be destroyed before **creator** exits the dialog during which they were created (if it was in one). Tasks created by elaboration of declarations are similar to processes created via a COBEGIN-COEND facility [Dijk68]. Their threads of control emanate from the same point along **creator**'s thread of control and they all coalesce at another point along **creator**'s thread of control.

Suppose **created** is declared in a declare block nested within **creator**'s DISCUSS statement, or in a procedure called by **creator** within **creator**'s DISCUSS statement. Then all of **created**'s actions over its lifetime must be recoverable. This means that any tasks with which **created** communicates must be participants in **creator**'s dialog. This is reasonable since such a task as **created** can be considered an extension of **creator**'s algorithm.

Suppose **created** is declared in **creator**'s declarative part. Then it must be destroyed before (or as) **creator** is destroyed. In this case, if **creator** can be backed up beyond its creation, so must **created**. **Created** is created within a dialog if **creator** is created within a dialog. Thus, **created**'s co-communicants must enter any dialog during which **creator** was created. We reserve an exception to this last statement for the case in which **creator** is created via the as-yet-unelaborated case of tasks created by an allocator in a statement sequence.

It is simple to define rules that would prevent smuggling in the declared task cases. In anticipation of more problems in the general case, we restrain ourselves from defining rules that might not generalize easily. We will apply to declared tasks special cases of a solution to the general case (below).

7.2.2. Allocated Tasks

We now consider tasks created via allocators. Tasks actually created via an allocator can survive their creators. Whether they do or not depends upon the position of the *declaration of the access type*. Although they might need to communicate with their creators initially, they might also need to survive any dialogs their creators might have been in during the creation activity; at least that is the *intent* of the facility.

Attempts at Solution

Three of the approaches one might consider making when defining of which dialog a newly created allocated task is a member are,

(1) **Created** is in some dialog upon creation. We have not yet provided a syntax capable of expressing **created**'s exit from that dialog.

(2) **Created** is outside any dialog and can enter any dialog it wishes when it wishes. If **creator** has to back up beyond **created**'s creation, **created** gets abandoned. This action could have *disastrous* consequences for whomever **created** is communicating with at the time!

(3) **Created** is outside any dialog and can try to enter dialogs, but gets suspended until **creator** can no longer be backed beyond the creation. Tasks created by tasks in the same dialog can communicate amongst themselves, but not with other tasks outside their creators' dialog.

We show below that each of these approaches is incomplete.

7.2.2.1. Consequences of Approach 1

Consider the first solution attempt outlined above. Suppose **created** is considered to be in a dialog upon its creation. This can be the same dialog as **creator**, or an entirely different one.

If **created** is in some dialog when it begins its existence, we do not have a (syntactic) way of expressing **created**'s normal exit from that dialog. In particular, we have no way of specifying a local acceptance test for **created**. A syntax must be found to enable **created** to contribute, via acceptance tests, to the error detection capabilities of the dialog.

To illustrate, **creator** might execute its allocator while outside of any dialog, from within any one dialog, or from within the inner of several nested dialogs. Somehow, the text of **created** must be able to specify an *undeterminable* number of acceptance tests to provide for successful exit of the dialog(s). The source code of **created** is static and, as such, can only be prepared for one of these possibilities. Either none, one, or some other specific number of tests can be there, but *not* an indeterminate number.

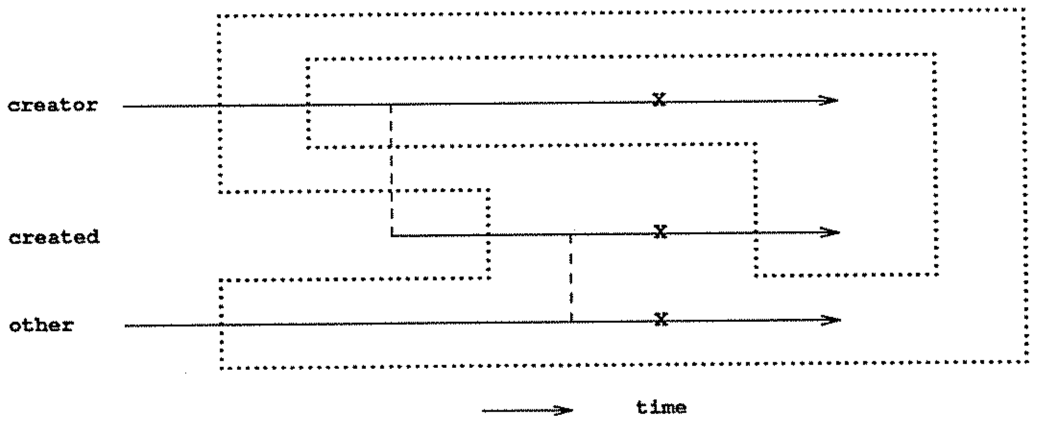
On the other hand, if **created** is considered to be in a different dialog from **creator**, not only do we need a way of expressing normal exit, but we also need to determine the consequences of failure of **creator**'s dialog. These consequences, as they are shown below, provide the basis for rejecting this solution attempt.

Suppose **created**'s dialog is not nested within **creator**'s. **Created** will be destroyed by **creator**'s backward recovery, so any other task in **created**'s dialog must be backed up to that dialog's recovery line. If this is not done, the other participants in **created**'s dialog would be caught up in a trap or retain (smuggled) knowledge of **created**'s existence. Also, if **created**'s dialog is not nested within **creator**'s, it might be possible for **created** and whoever else might be in that dialog to commit before **creator** even *starts* backward recovery. Should that happen, the other participant(s) in **created**'s dialog cannot be made to behave as if **created** had never existed. They would have discarded the recovery point(s) associated with **created**'s dialog, preventing their being backed up. This would constitute smuggling from **creator**'s dialog in that some task(s) outside it would have received information about happenings (attempted creation of another task) within it. This approach is missing semantic rules that would prevent **created**'s smuggling information about its existence out of **creator**'s dialog.

7.2.2.2. Consequences of Approach 2

Next, we consider the second solution attempt outlined above, that **created** is not considered to be a participant in any dialog upon its creation. This can evolve into the first approach in which **created** begins in a dialog other than **creator**'s. That is, **created** may simply enter its first dialog before **creator** exits all of its dialogs. This is an interesting point, but not the worst problem.

Suppose **created** and **creator** need to communicate explicitly, which might well be expected of them. If **created** had been considered to be in **creator**'s dialog, the two could communicate. However, if **created** does not start in any dialog, it needs to enter **creator**'s dialog to communicate with it. This would be acceptable if **creator** were in only one dialog, but if it were in several nested dialogs, **created** would have to "know" how many and which ones. That is only one problem. Suppose, as depicted in Figure 7.2 (solid time lines progress to the right, dialog boundaries are dotted, and communications are



A Consequence of Attempted Solution 2

Figure 7.2

dashed lines[†]), that **created** does enter **creator**'s nest of dialogs to communicate with **creator**. This affords the opportunity for **created** to communicate, on its way in, with some task which is a participant in one of the outer dialogs but not of the inner. After that point (marked by **x** in Figure 7.2), backing **creator** beyond the decision to create **created** would require undoing **created**'s communication with the participant of the outer dialog. That requires backing that participant out of its innermost dialog, which in turn requires that **creator** also be backed out to the same level. In general, **creator** would have to be backed out of its outermost dialog.

If we were to adopt the position that **created** is not in any dialog while **creator** is, we would retain the need for a solution to smuggling. We would also have added other potential causes of errors. We have shown that it would be possible for **creator** to be backed out of its outermost dialog if **created** had to be un-created due to failure of **creator**'s innermost dialog.

7.2.2.3. Consequences of Approach 3

Finally, we elaborate on the third solution attempt outlined above. If **created** exists as a result of execution of an allocator, it can survive **creator**. Thus it retains the syntactic problems mentioned above for exiting a dialog within which it might be created. Its place in the dependency graph and its potential lifespan do not naturally make **created** an extension of **creator**. It seems to make more sense to have such tasks begin execution as part of no dialog. Thus, we might choose a definition under which **created** is outside of any dialogs upon its creation. So far, this resembles the second choice with the same problems of needing to back out of communication if **creator** must back up. Instead, we might define a set of tasks created like **created** by tasks within the same dialog as **creator**. **Created** could communicate with any task in this set. **Created** would block upon attempting to enter a dialog with a task outside of this set. **Created** would remain blocked until **creator** discards the recovery point that endangers

[†] Were it not for the foregoing context, that the implicit initial communication between **creator** and **created** is satisfactorily protected, the fact that the leftmost dashed line crosses dialog boundaries would immediately alert us to a smuggling problem.

created's existence.

What follows then could be the additional semantics associated with creation of an allocated task. **Created** is outside any dialog and can try to enter dialogs, but, if it does, it gets suspended until **creator** cannot be backed beyond the creation. Tasks whose creators are in the same dialog can communicate amongst themselves, but not with anyone outside their creators' dialog. If **creator** backs out of its dialog, **created** ceases to exist. If **creator**'s dialog succeeds, **created**'s dialog gets released from protection.

This makes the dialog data structure of Chapter 4 only marginally more complicated. When **created** enters a dialog, a new `dialog_entry` is dynamically formed and is attached to **creator**'s current `dialog_entry`. This `dialog_entry` accounts for the dialog for the set of tasks with which **created** is allowed to communicate. It becomes attached to `dialog_entry`s of parent dialogs as **creator** (successfully) exits its nest of dialogs. The dynamic `dialog_entry` does not follow **creator** should **creator** once again descend deeper into dialog nesting. This upward migration continues until there is no parent dialog. At that point, the information in the `dialog_entry` is merged with the permanent `dialog_entry` with the same name.

Besides merging **created**'s floating `dialog_entry` with a permanent one as above, there are other options.

(1) All participants in one of the (permanent and floating) dialogs of the same name could be suspended until the other dialog ends. This option assumes the two dialogs were indeed intended to be unique. It also enables reuse of the static execution-time data structures of Chapter 4.

(2) A set of temporary `dialog_entry`s in the `dialogs_table` could be created to accommodate the participants in **created**'s dialog. This option also assumes the former protected and non-protected dialogs were indeed intended to be unique. It enables continued parallelism but requires a more complicated execution-time data structure.

Taking into account the dependency structure of allocated tasks has motivated what seems to be a solution. We have also offered three possible ways to deal with ongoing dialogs among created tasks when their creators' exit from their outermost dialog releases the created tasks from danger of destruction due to backward recovery. Unfortunately, this attempted solution ignores the fact that in many applications **creator** and **created** do need to perform some initializing communication. This is an important point. Taking it into consideration will lead us to an actual solution to the dynamic process creation problem.

7.2.3. Solution to Smuggling Due to Dynamic Task Creation

Implicit communication resulting from task creation is not handled by the `dialog_sequence` as defined in Chapter 3, Appendix A, Chapter 4, and Chapter 6. We present here a revision that constitutes a solution for the most general class of task, and we provide special cases of the solution for the other classes of tasks.

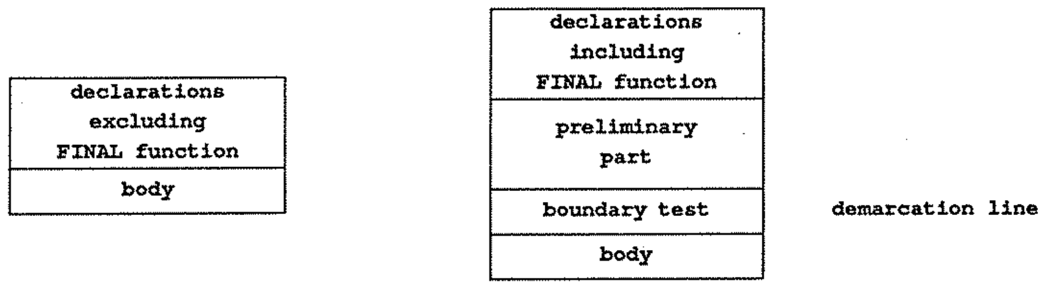
7.2.3.1. The Allocated (General) Case

A solution that pays attention to initializing communication does not need the options that the third partial solution of the previous section needed. Under that partial solution, new tasks were outside any dialog and could try to enter dialogs, but they would become suspended until the creating task could not be backed beyond their creation. Our realistic solution contains a compromise among the better parts of the choices presented above.

Tasks actually created by an allocator in a statement sequence can survive their creators. Thus, although they might need to communicate with their creators initially, they need to survive any dialogs their creators might have been in during the creation activity. This indicates a need for some demarcation within the text of **created**, between the initial recoverable activities during **creator**'s dialog and subsequent activities. At such a point, **created** could be blocked until the end of the dialog, and the associated determination by **creator** and its colleagues to commit to **created**'s existence.

A Preliminary Part

Figure 7.3 depicts the structure of a task as Ada exists now beside the structure of an allocated task in a language derived from Ada under this plan. Figure 7.4 depicts a possible syntax denoting that structure. There is a *preliminary part* which is executed while **creator** is still within the innermost dialog during which it created **created**. When **creator** reaches the acceptance test for that dialog, and **created** reaches the demarcation line between the preliminary part and the rest of its body, the boundary test serves as **created**'s local acceptance test for that dialog. There can be GOTOs out of the preliminary part, but such branches are delayed at the boundary test, just as in a DISCUSS statement. A task is not allowed to reenter the preliminary part. The FINAL function in **created** is a last chance test on **created**'s view of its status as **creator** evaluates the acceptance test on its outermost dialog. The FINAL function is evaluated asynchronously with the statements following the demarcation line in **created**'s body.


declarations excluding FINAL function
body
declarations including FINAL function
preliminary part
boundary test
body
demarcation line

Organization of Tasks That Can Be Created Via Allocators

Figure 7.3

```
TASK BODY <id> IS
   <declarations>
   FINAL FUNCTION <id> RETURN BOOLEAN IS
      ...
   END <id> ;
BEGIN
   PRE <statements> TO ARRANGE <boolean expression>
   <statements>
END [ <id> ] ;
```

Syntax for Allocated Tasks

Figure 7.4

After successful completion of **creator**'s innermost dialog and before completion of its outermost dialog, **created** "floats" out. This means **created** thinks it is at the outermost level (outside a dialog) and enters dialogs to communicate with others. These dialogs are actually nested within those of **creator** in a manner that will be explained shortly. The other tasks with which **created** can communicate in these dialogs are **creator**, other tasks created within **creator**'s dialog nest by **creator** or **creator**'s co-participants, and **creator**'s co-participants themselves. If a dialog's membership consists solely of created tasks, that dialog can continue while **creator** and its colleagues evaluate acceptance tests and step up in the dialog nest. If one of these dialogs has a member that is not in the same situation as **created**, **creator** and its colleagues must of course wait. This occurs because that member's DISCUSS statement for its dialog with **created** must be strictly nested within its DISCUSS statement for its dialog with **creator**.

Rationale For The Solution

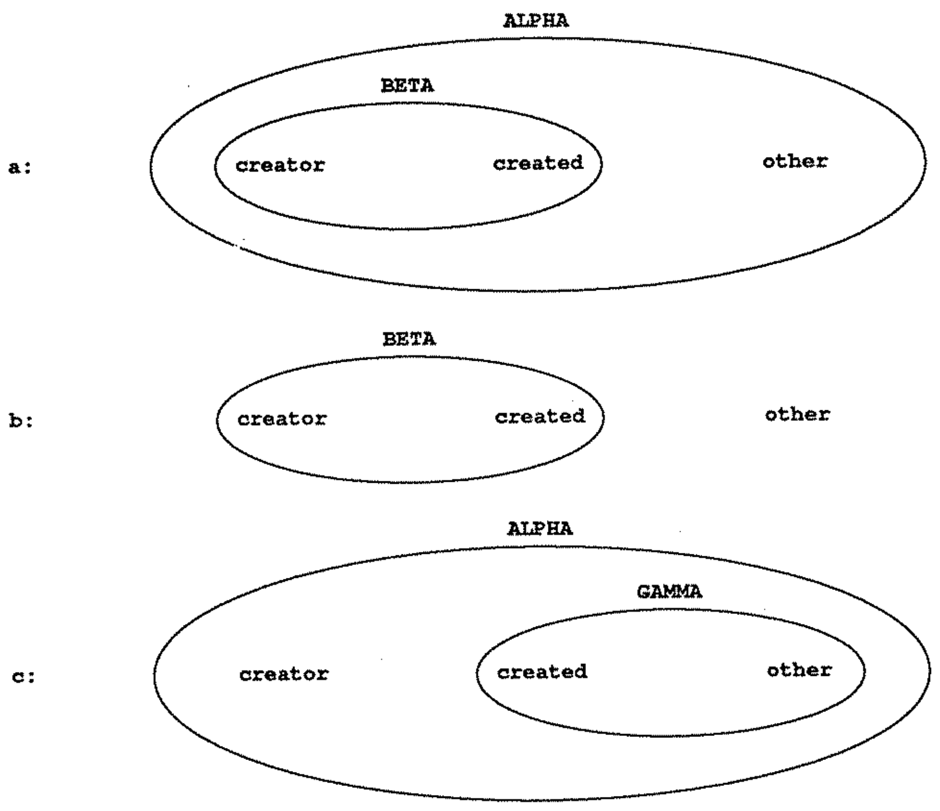
There are several questions one might raise in connection with this solution. Can **created** prevent progress by staying in its preliminary part? How can **created** communicate with **creator** if it "thinks" it is not in any dialog during the preliminary part? When do **creator** and **created** have to synchronize as they exit the dialog nest? With which tasks other than **creator** can **created** communicate? How does

created avoid needing to know for how many nested dialogs to provide acceptance tests? Why does not **created** simply lie dormant until **creator** exits its outermost dialog? We answer each of these in turn.

If **created** gets into an infinite loop or otherwise blocks in the preliminary part, there is no TIMEOUT to save it unless the TIMEOUT exists in **creator** as **creator** awaits **created**'s termination.

Created cannot communicate directly with **creator** while in its preliminary part. However, they *can* communicate if each of them enters a nested dialog. As illustrated in Figure 7.5a, from **creator**'s viewpoint, this dialog is nested within the dialog during which it created **created**. As shown in Figure 7.5b, **created** views this dialog as a top level dialog, although it is actually nested within the dialog of **creator** during which **created** was created. This particular structure has a side benefit of allowing **creator** to try several times, if necessary, to get **created** started properly without backing beyond the creation itself. It also provides a path, shown in Figure 7.5c, through which some other participant in **creator**'s dialog may communicate with the preliminary part of **created**.

Suppose, as in part a of Figure 7.6, that **creator** were in dialog BETA nested within dialog ALPHA when it created **created**. When **creator** reaches the end of BETA successfully, **created** might be at the demarcation line or somewhere inside the preliminary part. If **created** reaches the demarcation line first (Figure 7.6b), **created** (unless it is being terminated) cannot proceed until **creator** has successfully exited BETA. On the other hand, if **created** is somewhere inside the preliminary part (Figure 7.6c), **creator** has to wait until **created** successfully evaluates the boundary test. TIMEOUT's in **creator** can break up the situation if **created** is somehow blocked. **Creator** and **created** are not similarly blocked at the end of **creator**'s outermost dialog, ALPHA in Figure 7.6d. There, after **creator** has successfully evaluated its acceptance test, **created**'s FINAL function is evaluated asynchronously to **created**. If **created** reaches its end before the end of ALPHA (Figure 7.6e), **created** terminates, but its FINAL function is still evaluated when **creator** reaches the end of ALPHA.

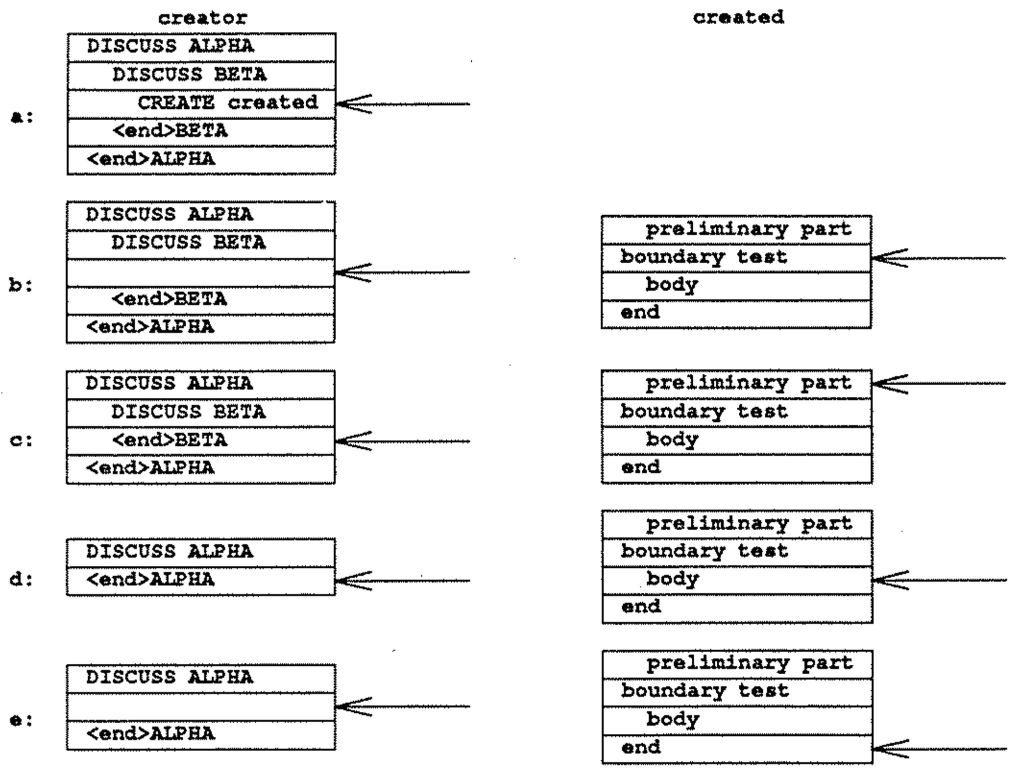


Views of Dialogs Within a Preliminary Part

Figure 7.5

Within the preliminary part, **created** can communicate with another task in **creator's** innermost dialog or another similarly created task. After that, the set of tasks with which **created** can communicate gradually expands as **creator** exits more and more dialogs. The additional tasks at any level are the participants of that dialog that did not enter the lower level with **creator** and the new tasks similar to **created** for which those tasks are responsible.

There are not multiple preliminary parts of **created** for the case in which **creator** is in nested dialogs. After the preliminary part, **created** is in whichever dialog **creator** happens to be until **creator** exits its outermost dialog. **Created** is in a special state between leaving the preliminary part and



Relative Progress of Creating and Created Tasks

Figure 7.6

evaluating the FINAL function. It is within **creator**'s dialogs, yet *does not have to participate in acceptance tests* whose numbers it cannot know. Nor does it really obey the other nesting rules except at the topmost level of **creator**'s dialog nesting.

This solution has addressed the questions of whether there should be an acceptance test at the demarcation line, and if so, whether it is related to the innermost or the outermost dialog, or to each along the way. One really does not want a newly created process to be trapped in its creator's set of nested dialogs. Preliminary code for communicating with the creator is fine at the innermost level, but at uncounted intermediate levels the programmer would begin to get lost. Part of the reason for this indeterminate or unbounded dialog participation after the preliminary part is that one might not know how deep the nesting is for any instantiation of the new task. To have a newly created process communicate only with the creator and only at the innermost level then wait until the creator cannot uncreate it is a *major* restriction. A communicant with the creator might need to communicate with the new process on the way out of the nest. Indeed, Figure 7.7 shows that the scope of the access type (if created via an allocator) may disappear before the creator finally exits the outermost dialog. For example, after execution of line 13, a task of type BTA exists. We have given rules that allow that task to communicate with task B.

The preliminary part and FINAL function constitute a solution to smuggling as a result of dynamic process creation. Its exposition has been in terms of the allocated task case. We next show how it is

```
 1: task B is ... end B;
 2: task A is ...
 3:    discuss ALPHA by ...
 4:       declare
 5:          task type BTA;
 6:          type PTR is access BTA;
 7:       begin ...
 8:          discuss BETA by
 9:             declare
10:                X : PTR;
11:             begin
12:                discuss GAMMA by ...
13:                   X := new PTR; ...
14:                to arrange ...
15:             end;
16:          to arrange ...
17:       end;
18:    to arrange ...
19: end A;
```

Process Creation via an Allocator

Figure 7.7

applied to the special case of declared tasks.

7.2.3.2. The Special Cases

The tidy lifetimes of tasks created in declarations are the exception rather than the rule. For these tasks, the dependency rules imply that **creator** cannot exit the dialog in which it is created before **created** is destroyed. We model the activities of declared tasks as if they were allocated tasks, with an empty statement sequence beyond the preliminary part. This means the entirety of a declared task's activities occur within its preliminary part. If **created** is created as a result of elaboration of the declarative part of **creator**'s body, then **created** is in the same dialog as is **creator** upon **creator**'s creation. This should be checkable and therefore enforceable at compile-time.

For a task created as a result of elaboration of declarations, the syntactic nesting structures of the language require that **created** must be destroyed before **creator** leaves its dialog. This means that we do not need a syntactic expression of **created**'s normal exit from the dialog if we define **created** to start in **creator**'s dialog and spend its ''life'' in the preliminary part. As for **created**'s communication with tasks that are not in **creator**'s dialog, **created** can be considered to be an extension of **creator**; thus, the other tasks would need to enter **creator**'s dialog to communicate with **created**. Further, the visibility rules ensure that **created** must initiate such communication; outsiders cannot ''see'' **created**. Because of the method of creation, no outside process can see **created** to attempt communication with it. However, **created** can communicate with tasks on the outside, so the execution-time check can be made. **Created** can communicate with any task within the same dialog as **creator**, or within a nested dialog if **created** explicitly enters one.

This provides a solution for declared tasks. As declared tasks are special cases of allocated tasks, the solution for declared tasks is a special case of the solution for allocated tasks.

Task creation in Ada is one specific example of the general case of dynamic process creation. We have shown the problems this facility presents for inclusion of backward error recovery in a language. We have also shown modifications to the semantic rules of the dialog and colloquy concepts that effectively deal with these problems without severely restricting the degree of parallelism.

We have not given a different syntax for declared tasks. Whether we use Ada's syntax or the new one created for the allocated case, the programmer (designer) must program declared and allocated tasks differently. To have the programmer reminded of that requirement by syntactic differences or by different placement of the bulk of executable statements within the body could even be beneficial in preventing faults.

7.3. Process Destruction

Process creation is only one form of process manipulation. Another is dynamic process destruction. Again, we illustrate with Ada tasks. Like task creation, task destruction is a form of communication. Here, we examine it for similar pitfalls.

Tasks can be destroyed due to their normal control flow or through the effects of ABORT statements. Implicit communication resulting from any form of task destruction is not handled by the `dialog_sequence` as described so far. We will show that normal termination is not a problem. Aborted tasks and their implications for task attributes embodies the bulk of this section and carries over into the next.

7.3.1. Normal Termination

Normal termination of tasks does not interfere with the inclusion of dialogs. Normal termination of tasks in Ada cannot engender smuggling. The syntactic nesting requires that a task exit any dialogs it had ever entered before it reaches its END. This is so even in the situations requiring the use of algorithm BRANCH in Chapter 4. Failure of a test along the way indicates that something, perhaps including the decision to take the "current" control transfer, was wrong and must be undone. Control transfers due to the raising of exceptions have been integrated into the definitions of the DISCUSS statement and `dialog_sequence` already.

A terminate alternative in a selective wait within a DISCUSS statement can be handled by suspending the task until one of three things happens. These are,

(1) one of the open entries is called,

(2) the dialog fails, or

(3) all other participants in the dialog reach their acceptance tests (or their own terminate arms).

When all other participants have reached their acceptance tests, there will never be any other task that can enqueue a call on the unique queues established for that dialog. At that point, the task evaluates its own acceptance test. If the dialog fails, the task will be backed up to a point before it began to try to terminate. If the dialog succeeds, the situation is as if the dialog sequence were replaced by a selective wait with terminate alternative and no other open alternatives. Under this approach, the task is seen to ease out of its nest of dialogs, with implied GOTOs directly to acceptance tests where it must wait for other participants of the respective dialogs. Failure of the dialog that had the terminate arm means the task is not trying to terminate, rather it is participating in the next (alternate) dialog. Message arrival can only back the task out of the try for termination inside the innermost dialog.

7.3.2. Aborts

Aborts are a much more serious problem. It is clear that this kind of task destruction is a form of communication. A programmer writes the statements following an ABORT statement with the knowledge that the tasks that were the objects of the ABORT statement no longer exist, or that they will cease to exist upon their next attempt at synchronization. The programmer knows something definite about those tasks' states, and cannot help including that knowledge in the subsequent code.

This communication is one-way, in that only the executor of the ABORT statement gains information and the object of the ABORT is neither a willing nor knowledgeable participant. It is the fact that the object of the ABORT statement is not *knowledgeable* that prevents us from requiring the two tasks to enter a dialog for that communication. Further, because the object is a "moving target", we cannot require the task executing the ABORT statement to enter whichever dialog its target happens to be in (that is, to catch up to the target) as a prerequisite to executing the ABORT.

A task that is aborted can seem to other participants in any dialog in which it happens to be as if it were suffering from faults that might generate `tasking_error` upon attempted communication. The aborted task will not reach its local acceptance test allowing the dialog to end. If the dialog ends with failure, the aborted task must be removed rather than backed up (unless the aborting task was also in that dialog). Once backed up, the other tasks may try to re-enter a dialog with the aborted task, but will find that it has not deigned to enter with them. If the dialog is to end with success, the exit algorithm must be modified to allow for more tasks to enter a dialog than try to execute acceptance tests (i.e. the abort code must also manipulate the set of participants).

Another important problem with task abortion and backward error recovery does not involve smuggling so much as implementability. The aborted task must be resumed if the aborting task is backed up beyond the decision to abort. In order to be able to resume it, an implementation must save *all* of the aborted task's resources until the aborting task commits so that the aborted task can be reenstated if the aborting task is backed up.

Suppose a task has an entry call queued on an entry of the task that is to be aborted. The calling task need not be in the same dialog as the targeted task. The Ada LRM requires that the calling task receive TASKING_ERROR and be removed from the queue. This is to happen even if the calling task initiates the call after the ABORT. If the calling task later tries to call again and does not receive TASKING_ERROR, it can conclude that the first rendezvous began and an exception was raised in the callee or that the callee was the target of an ABORT that was later rescinded. If the calling task can reach this latter conclusion, smuggling has taken place. That it is possible for another task to reach that conclusion is shown in the next section.

The remaining and major problem with aborts relates to smuggling through task attributes between the time that the ABORT statement is executed and the time that the destruction of the targeted task becomes permanent. Examining that manipulation will help us to understand this one enough to seek a solution.

7.4. Task Attributes

In programming languages, like Ada, that provide an unrestricted shared variable facility, one process may examine the internal *data* state of another. The restrictions that were presented in Chapter 6, deal with the problems that are raised by this unrestricted access. There are also facilities in some programming languages through which one process may examine the *execution* state or progress of another. Such facilities exist in Ada and are known as task attributes. When a process is backed out of an execution state that has been examined by another process, the information is invalidated. Thus, to provide satisfactory backward error recovery, either both must be backed up, or the information transfer must be prevented.

The `T'terminated` and `T'callable` attributes in Ada are used by tasks to gather information about the internal states of other tasks, so they seem to behave like shared variables. Their actual use indicates that they are not. If the process using these attributes is in a dialog at the time, and the dialog fails, the information is lost; otherwise the situation is the same as if the asker had not been in a dialog. A task cannot be backed out of the terminated or un-callable conditions unless they arose due to an ABORT statement within some dialog. If the task is created within a dialog and terminates before exit, no outside task can reference that task to ask these questions.

These attributes are supposed to behave as sticky bits. They start off as FALSE (TRUE for `T'callable`), and once TRUE (FALSE), they stay TRUE (FALSE). The situations to be examined are those in which backward recovery might change them from TRUE (FALSE) to FALSE (TRUE). As we showed above, under conditions of normal termination, the subject cannot be backed up by failure of a dialog. This narrows the scope of the question down to aborted tasks. Notice from the previous section that whether the result of an entry call is a TASKING_ERROR can be used to examine the callee's execution state (status). This and the narrowed scope of the task attribute problem is why we put off a solution to task abortion in the previous section. The combination of task attributes and task abortion is discussed below.

7.4.1. The Problem with Task Attributes

There are some subtle meanings of the combinations of `T'terminated` and `T'callable` with the ABORT statement. In discussing these issues, there are three tasks of concern.

(1) The killer (K) is the task that executed the ABORT statement.

(2) The target (T) is the task that is the object of the ABORT statement and of the inquiries.

(3) The asker (A) is the task making these inquiries.

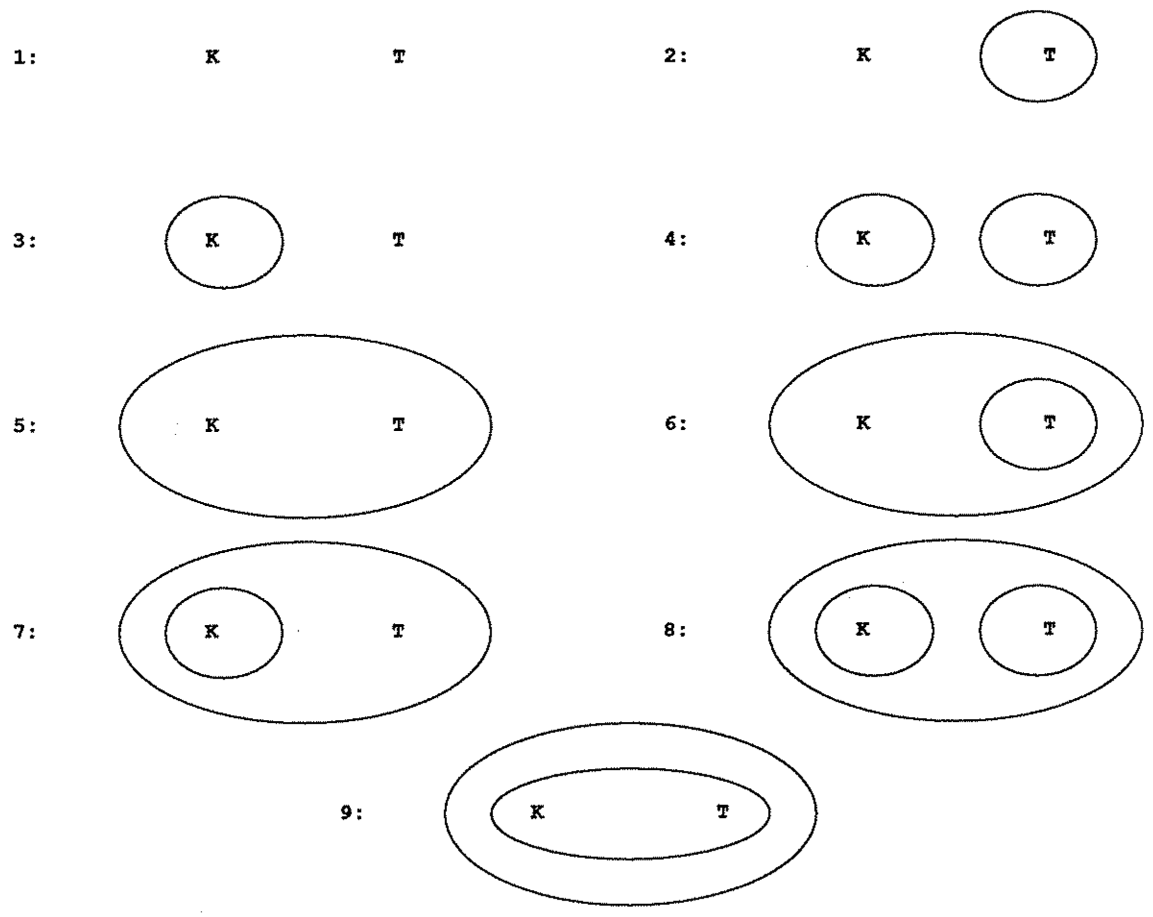
There are a number of cases depending upon how deeply and within which dialogs the three tasks happen to be nested when the ABORT is executed and when the inquiry is made.

If the result of the inquiry (FALSE for T'terminated and TRUE for T'callable) indicates that T has not been aborted or otherwise terminated, There is no problem. If A is backed beyond the inquiry, these values remain unchanged. The problem with the information acquired by A lies in the cases wherein K is backed up but not A. In such cases, we must have some mechanism for reversing any decisions that A might have made based upon the invalid information, or for suspending A's progress until such time as the information becomes permanent.

We have discussed most situations in which T might be backed up. In those cases, T remained terminated. However, if K is backed up with T, T should not remain terminated. Indeed, if K is backed up independently of T, T must become "un-terminated", and T's companions must be backed up through any interactions that they may have attempted with T.

Thus, everything revolves around recovery of K. K is the task that made the decision to execute the ABORT statement, K is the task whose decision could be reversed. That reversal invalidates information acquired by A about T.

Figure 7.8 shows nine of the cases of dialog nesting for K and T at execution of the ABORT statement. A bubble represents a dialog with its participants listed inside. In cases 1 and 2, K cannot be backed up, so we do not have to worry about un-terminating T. In cases 5 and 6, if K backs up, so do T and its fellows. In case 4, T and its fellows must not be allowed out of their dialog (must not throw away their recovery line) until K has successfully completed its dialog. Case 8 resembles case 4 but mutates into one of case 5, case 6 or case 7. In case 7, we have a choice of blocking T and any other tasks' activities involving T until case 7 mutates into case 5, or of having any backup of (K's) inner dialog immediately trigger backup of the outer dialog. Case 9 and any further nested cases are extensions of the first 8 cases.



Dialog Nesting Situations for K and T Upon ABORT

Figure 7.8

This leaves case 3. Notice the similarity between case 3 and case 7. For case 7, we had a chance to block any operation involving T until K committed. The alternative, which is inapplicable to case 3, was to permit operations involving T, but to back them up as well as K should K's dialog fail. The choice is inapplicable because there is no further dialog surrounding both K and T that would allow them to be backed up together. To block all operations involving T means that the aborting activity and any other activities involving T must await the completion of K's dialog.

The only easy cases above were those in which T was in a dialog that was within that of K, or in which K was not in a dialog. The others involved blocking activities of other tasks until K arrived at a position relative to T that mirrored the easy cases.

If we substitute "A" for "T" in each case in Figure 7.8, we find the cases for the task executing the ABORT statement and the task making the inquiries about the aborted task at the time of inquiry. It is important to notice that it does not matter to the inquiring task where the aborted task is. What does matter is whether the decision to abort can be reversed. An implementation of these inquiries must determine whether to return an immediate answer or to block the asker (A). This determination must be based upon the dialog nesting status of the killer task (K). Thus, the implementation must inquire of K's task control block without knowing beforehand which task K is. Presumably that information could be found through the target (T).

7.4.2. Unsatisfactory Approaches

A might use the result of the attribute inquiry in an acceptance test and make a wrong decision based on K's wrong decision. This is indirect inter-dialog communication. Hence, we might consider blocking A until the result becomes permanent. Unfortunately, this decision introduces the possibility of deadlock.

To see this, suppose we combine case 5 (really case 9) for A and K with case 8 for T and K (Figure 7.9a). If A is blocked until a definite, permanent answer to the inquiry can be discerned, this will mutate into a combination of A and K case 5 and T and K case 7 (Figure 7.9b). At this point, the system of three tasks *deadlocks*. A cannot proceed until K exits its inner dialog, and K cannot (successfully) exit its inner dialog until A unblocks. The situation would almost be better if a third value (perhaps denoted maybe) could be returned by the attributes.

7.4.3. Practical Approach

The "easy" cases were those for which the decision to abort was permanent. Indicating that abnormal task destruction may best be made an unrecoverable activity. As an unrecoverable activity, task abortion should fall into cases 1 and 2 only. Yet as a form of communication, it should fall into cases 5 and 9 only. Recall that task abortion is unlikely to fall into cases 5 and 9 as it entails K chasing T down. If task abortion is indeed to be an unrecoverable activity, a language designer might then wish to require special permission or authority for a task to wield such power.

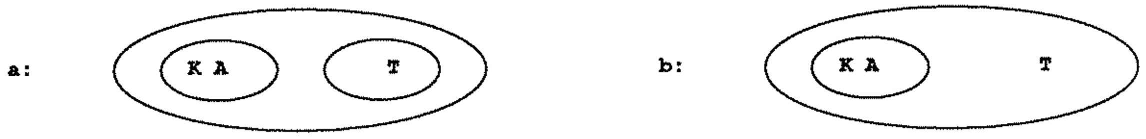
For purposes of the preceding paragraph, an activity that is unrecoverable

(1) is only done outside a dialog, or

(2) has no effect until its executor successfully exits all dialogs, or

(3) has immediate effect without possibility of backup.

Of the three, only the first does not allow a programmer to "forget" that the activity is unrecoverable. The second changes the "immediate effect" semantics of the Ada ABORT. The third can result in an inconsistent state due to partial recovery. Thus, we require that an ABORT statement never be nested



Deadlock Due to Attribute Inquiry

Figure 7.9

within a DISCUSS statement.

Abnormal process destruction causes problems for backward error recovery when we try to make that operation recoverable. Task abortion, with the foreboding warnings in the Ada LRM, seems to be intended as a form of forward error recovery. Rather than being a mechanism for backing up a task's progress, the ABORT statement destroys the targeted task. Only as a recognized unrecoverable activity does the task abortion facility cease to interfere with backward recovery.

Making task abortion unrecoverable has solved (or avoided) the problems of mixing backward error recovery with abnormal process destruction and external observation of processes' execution states as they occur in Ada. We emphasize that this is not necessarily a solution in the general case of external observation of processes' execution states.

7.5. Summary

We have described the problem of smuggling via unexpected information flows provided by existing facilities in a language designed for implementing crucial systems. One unexpected source of information flows is the manipulation of processes. We have examined three mechanisms for process manipulation and shown how they permit smuggling when backward error recovery is naively combined with them.

Smuggling via dynamic process creation can be prevented when the newly created process' initial activities are constrained within a preliminary part and checked by a FINAL function.

Dynamic process destruction and Ada's forms of examination of processes' execution states engender smuggling only in the context of abnormal termination (in Ada, the ABORT statement). The first of two possible solutions, blocking the process attempting to examine an aborted process' state until the abortion cannot be undone, presents additional problems and may even permit situations leading to deadlock. The other solution, forcing execution of the ABORT statement to occur outside of any dialog, is disquieting for so important an activity.

As with expected mechanisms for smuggling (Chapter 6), we have only dealt with a few unexpected mechanisms for smuggling. There may be others, if not in Ada, then in other languages. Also, as with our solutions for the expected mechanisms, our solutions for the unexpected mechanisms are often fragile and less than satisfying, as well as being defined using the semantics of a single modern language as an example.

Again, all of this shows that backward error recovery and these facilities in combination present problems not previously addressed or even acknowledged in the literature. It is clear that backward error recovery cannot be included in a programming language after the fact. It must be a design consideration from the outset.

CHAPTER 8

Program Structuring Problems

Recall that Chapter 5 described two major problems in merging backward error recovery into a realistic programming language. The first problem had to do with smuggling, and was dealt with from two standpoints in Chapter 6 and Chapter 7. The second problem is the incompatibility between deliberate establishment of recovery lines and the communications philosophy of the language.

This problem arises because the establishment of recovery lines must be a coordinated activity, agreed upon explicitly by *all* the participants. The facilities of most programming languages, however, are designed to avoid this level of explicit knowledge in communication structures. In Ada, for example, naming is one way only, and, in principle, an Ada task defining an entry does not know which other task is calling the entry during a particular rendezvous. This incompatibility in requirements leads to a general difficulty in program structure. We discuss here only a particular example, that of providing *service processes*.

To make this incompatibility clear, we first review Ada's communications philosophy. Next, we introduce some generalizations about the behaviors produced by this incompatibility. This is followed by some examples illustrating these behaviors, after which appear some solution approaches. Finally, we present a direction that might prove fruitful in a search for a comprehensive solution.

8.1. Background

Abstraction, Information Hiding and Modularity

Abstraction, information hiding and modularity have become "buzzwords" but are rooted in Parnas' original paper [Parn72]. We remind ourselves of the common threads.

Systems are to be divided up as re-usable, replaceable modules. For such a module to be useful to many users, or replaceable without modifications to its users, its specifications must completely abstract a general-case entity. Each module is implemented without knowledge of how many users there will be or which modules they are. The module is thus made re-usable. The users of the module know only the name of an instance of the provided abstraction and its specifications. Any knowledge of how the exported specifications are implemented is hidden from the users. The module's implementation is thus easily replaceable.

That a module does not know which or how many other modules are its clients implies one-way naming in communication.

As Anderson shows [Ande78], all this poses no problem for backward error recovery *in a sequential system*. However, as we saw with shared variables, problems arise when we expand to a concurrent system. Consider an abstract type. An instance is an object, which can be hidden as a package or as a task. A package is suitable for a sequential system and for a parallel system if each process has its own instance. If processes share an instance, we have either the confusion of simultaneous update of a shared object, or the shared object is guarded by a service process.

Service Processes

A *service process* provides a service for other processes. It is like a procedure that can only be called in a mutually exclusive manner, but that maintains an internal state between calls. With one-way naming, the service process does not know for whom it is providing the service at any time.

The use of service processes is the preferred method in many programming languages for sharing either simple variables or abstractions among multiple processes. The facilities of a service process are used when other processes need some operation performed on the abstraction. The service process often makes non-deterministic choices among users awaiting its various services. The services are performed during communication with the individual users, for example during rendezvous in Ada. These statements apply equally to monitors.

Ada "encourages" protecting shared variables in service (i.e. monitor-like) processes. The Ada (as a program design language) philosophy about shared variables seems to be to package them with a protective task [Cohe85]. As we show below, serious problems arise for backward recovery in these protective tasks when serving multiple clients.

Non-Deterministic, Exclusive Communication

The Ada language has nondeterminism in its communication mechanism. The dialog concept does not entail that kind of nondeterminism. Any quality that dialogs have which might be viewed as nondeterminism lies in the fact that a process cannot know which or how many other processes are using the same dialog name at the same time. The problem also involves a lack of exclusivity in dialogs. In our efforts to make the participants in dialogs anonymous, we have not provided for any form of mutual exclusion. The dialog envelope around communication has neither nondeterminism nor mutual exclusion. Due to this, the envelope restricts severely one's use of these qualities. Hence, it restricts severely the kinds of programs one may write.

8.2. General Behaviors Illustrating the Process Structuring Problem

We have identified several generalizations about the form of this incompatibility in terms of process interactions. We have named some of them the "Determinism", "Capture", and "Torpedo"

Effects. Others indicate that the designation of dialogs could require two-way naming, which in turn would run counter to modern design and programming precepts such as modularity, information hiding, and abstraction.

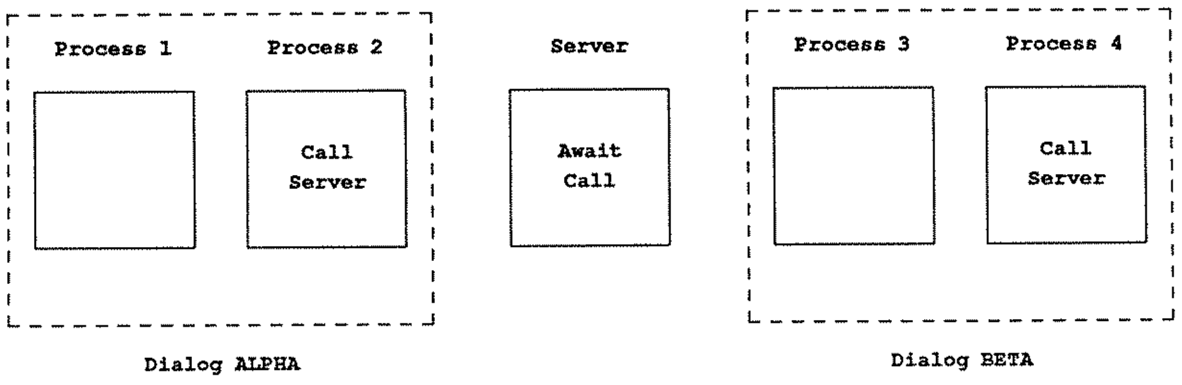
A server task may be requested to perform its service at any time in Ada. Under the dialog regimen, it seems a server must actively seek out its clients to achieve the same dialog nesting. To do so involves knowing its user's names to some extent. Further, the server cannot leave a dialog after dealing with one client and before seeking the next client until the first client is ready to leave.

Determinism Effect

The *Determinism Effect* occurs when a process cannot ascertain which dialog it should enter without entering one.

For service processes, the Determinism Effect is almost always in force. In order to meet its clients to provide its service, each server must know the names of the dialogs its clients are in at any instant and, for each client, the current proper depth of dialog nesting.

Consider a programming language that makes provision for both service processes and backward error recovery. A process, the *client*, that requires the use of a service process, the *server*, must communicate with the server. However, this communication *must* take place within a dialog and this requires an explicit action by the server. The server must enter a dialog in order to provide a service to the client, yet it cannot know which dialog to enter, or when, because it has no knowledge of the state of those processes that might be its clients. It doesn't even know their names. Indeed, if separately programmed, the server might not know at any given point which dialogs to enter in order even to be at the same level of nesting as a valid caller. This is the Determinism Effect. An example is shown in Figure 8.1 in which a server process, outside of all dialogs is idle, yet two client processes are unable to obtain service since each is in a separate dialog.



Inaccessibility Of A Server Process

Figure 8.1

Capture Effect

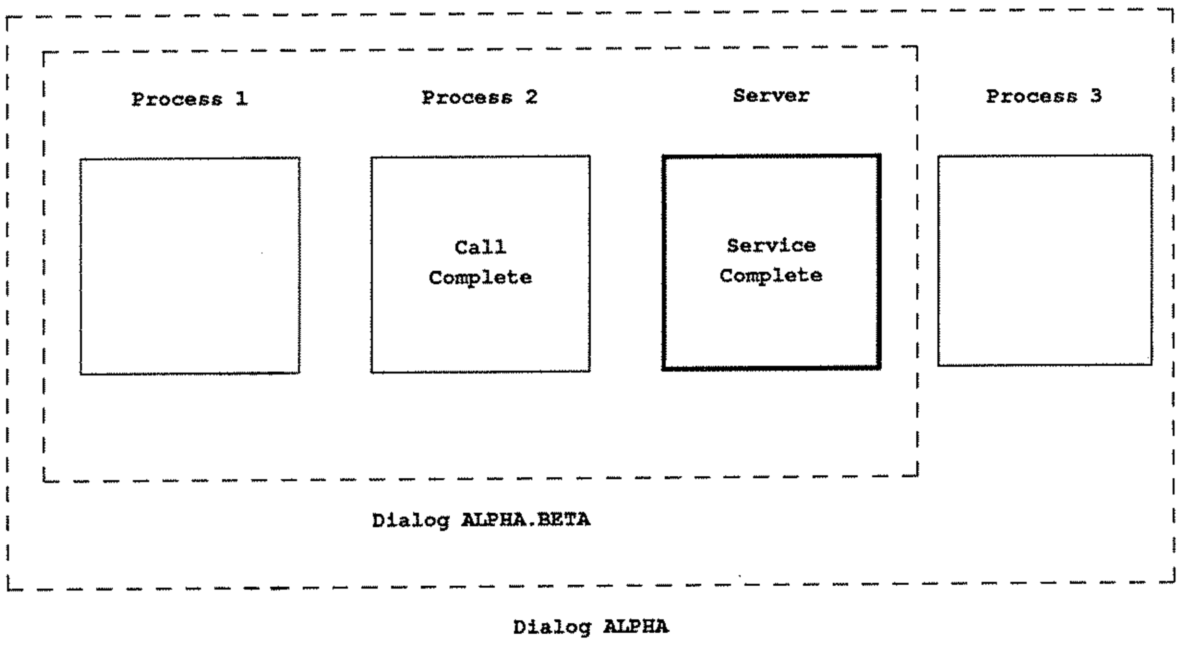
> The *Capture Effect* occurs when a process enters a dialog and is unable to exit until some other process is ready.

The captured process may need service whereas no service is available within that dialog, or it may have performed some service and be delayed in proceeding to its next client.

As a solution to the Determinism Effect one might naively consider arranging for service processes to be active, searching for clients, rather than passive, waiting for clients. A service process might be expected to enter a dialog merely to check to see whether its services are needed. Upon entry, the service process will have to establish a recovery point. Unfortunately, if its services are not needed, it has to await the completion of the dialog before it can continue since it has joined a coordinated set and must discard its recovery point with the rest of the set. This imposes an arbitrary delay on the service process. If its services are required it can provide them but then it still has to await completion of the entire dialog before it can leave and look for other clients. If there is a nest of dialogs, the service process will have to enter them all but await the completion of each before it can exit. In effect, by actively looking for clients, the service process will be "trapped" in every dialog that it enters until the other processes in the dialog determine that the dialog is complete. This phenomenon of entrapment until the dialog completes is the Capture Effect. An example is shown in Figure 8.2. The service task is nested within two dialogs but has no further clients. It cannot leave, however, until both dialogs complete.

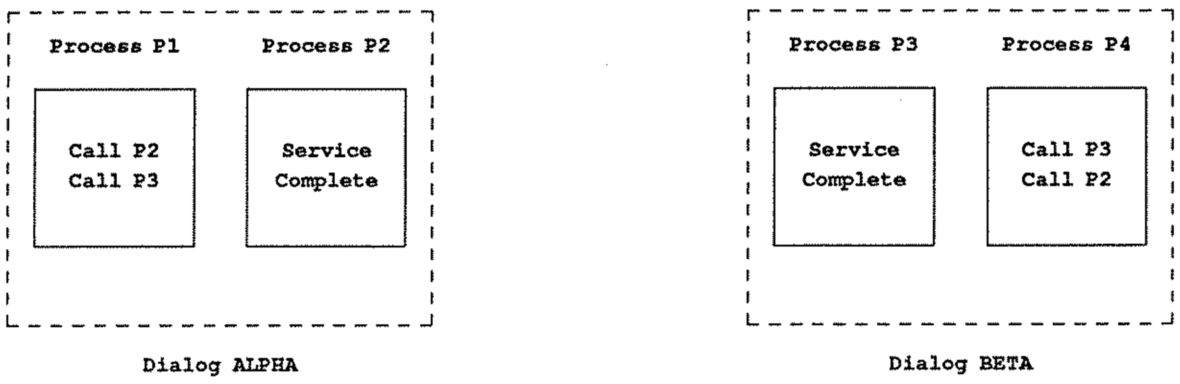
Deadlocked Resources

The Capture Effect can also lead to deadlock. Two processes that have provided their services and are ready to continue might be trapped in dialogs (Figure 8.3). If each dialog requires the attention of the other service process in order to complete, the system will be deadlocked. These processes' services are shared resources just as are shared variables. This indicates that the service process paradigm does not



Service Process Entrapment

Figure 8.2



Service Process Deadlock

Figure 8.3

mix with recovery lines.

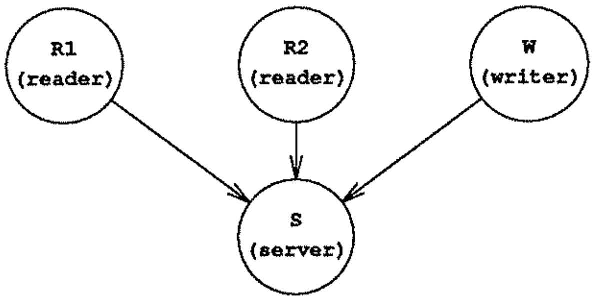
Torpedo Effect

> The *Torpedo Effect* occurs when a process enters a dialog and, by reason of the manner in which it is programmed to interact with dialogs, causes that dialog to fail.

Consider a process which, in attempting to overcome the Capture and Determinism Effects, uses TIMEOUTS to cycle through a list of dialogs that its clients may from time to time enter. If it cycles too quickly, it may flit from one client's dialog to another without ever serving anyone successfully. This is one variation of the Torpedo Effect. For another variation, consider a process that enters an existing dialog and causes it to fail, even though it otherwise might have completed successfully, hence backing up all participants. All participants are required to fail because of this process. In this variant, the process may have entered the wrong dialog; nevertheless, its interference undoes the others' accomplishments.

We give several illustrations of the Torpedo effect using the example shown in Figure 8.4. Suppose a service process (S) guards a buffer between a writer process (W) and two reader processes (R1 and R2).

Suppose S has a datum in its buffer. It has no way of determining whether next to enter the dialog meant for the readers or the dialog meant for the writer. If it simply tries one after the other, timing out when it finds no partners, a partner may have to wait unnecessarily in the second dialog while S waits in the first. This is the contrivance illustrated in the robot example program of Appendix C. S has no way to nondeterministically choose between dialogs with W and the R's. Again, we see the Determinism Effect at work. However, if S times out too soon, the situation worsens. S might arrive in R1's inner dialog (call it BETA), timeout, and timeout on an outer dialog (call it GAMMA) just as R1 and any other participants with it in dialog GAMMA complete an involved interaction. That would cause R1 and its companions to back up due to no fault of their own. Indeed, if S cycles through its list of clients' dialogs relatively quickly, R1 and its fellows might never accomplish the goals for which they entered dialog GAMMA. This produces the Torpedo Effect.



Buffer Service Task, One Writer, Two Readers

Figure 8.4

Earlier, we considered the server to seek out clients in their dialogs. Turning the strategy around, we might have clients join a server in a dialog of its choosing. In this case, we find that S cannot exclude all but one reader during the (read) operation. Suppose R1 enters a dialog with S for the purpose of reading a datum. If R2 enters the dialog before R1 and S disengage, R2's presence could jeopardize a successful exit by S and R1. This means the dialog would fail through no fault of S or of R1, but due to R2's normal and correct activities — another instance of the Torpedo Effect.

Two-Way Naming

Continuing to use Figure 8.4, suppose S has one datum which both R1 and R2 attempt to read. Suppose also that an attempt has been made to avoid the exclusion side of the Torpedo Effect by requiring R1 and R2 to use different dialog names. The logical extension of this solution is to require every client of S to use its own dialog name. This is equivalent to S knowing each client by name. That invalidates a major benefit of service processes; that is that the server is programmed without knowledge of its clients. Thus, the language's one-way naming is changed to two-way naming.

Let it be the case, as in Figure 8.5, that R1 requires a datum while in a dialog ALPHA, and again

```
R1: ...                                            S: ...
    [ALPHA ... S.e ... ]                              [ALPHA accept e ]
    ...                                               ...
    [GAMMA ...                                        [GAMMA [BETA accept e ] ]
              [BETA ... S.e ... ]                     ...
                                   ... ]
    ...
              ( [ and ] begin and end DISCUSS statements)
```

Skeleton Source Showing Determinism Effect

Figure 8.5

while in a dialog BETA within dialog GAMMA. S must sometimes enter dialog ALPHA to find out if R1 is ready to read the datum, and must at other times enter dialog GAMMA then dialog BETA to find out the same thing. Thus, S must know not one, but *several* dialog names for each client process.

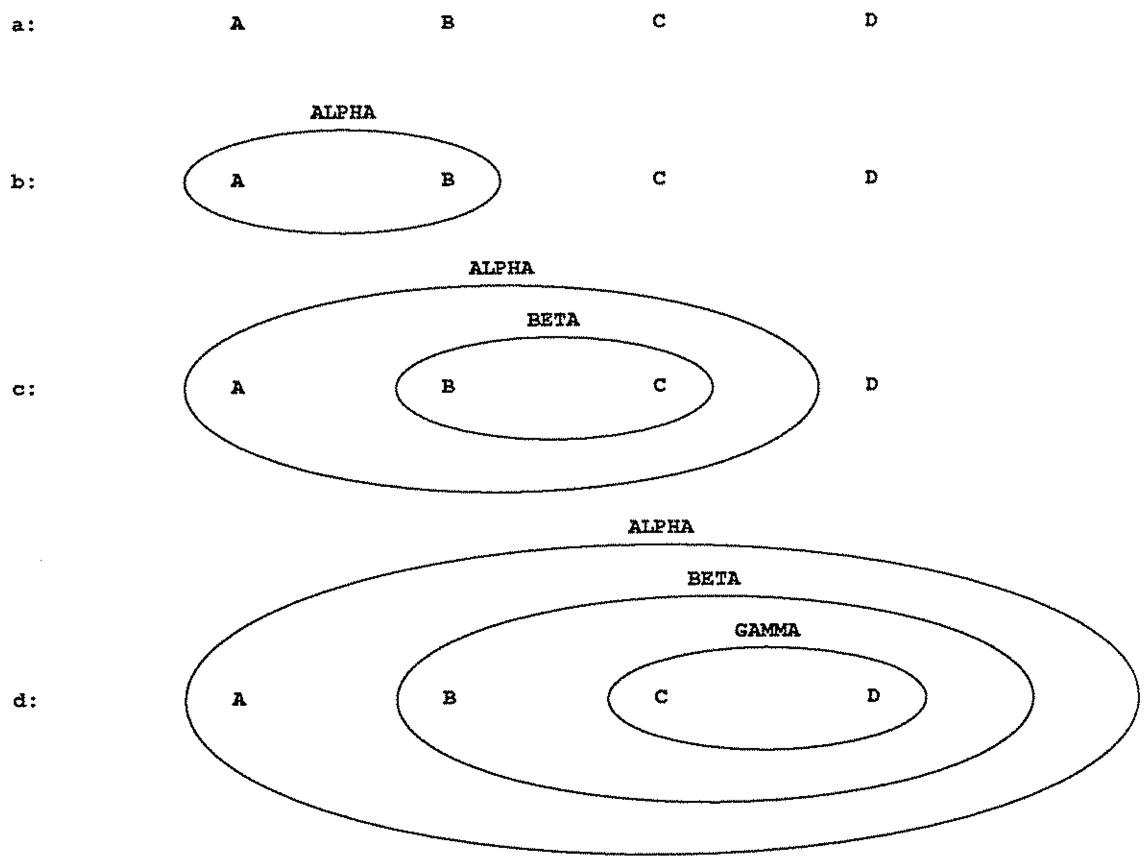
Counter to Modularity

Consider a system arranged as a hierarchy of objects where the objects are guarded by processes. Figure 8.6a shows such a system. One process (A) uses the object provided by another process (B). B implements its object in part by using the object provided by a third process (C). C implements its object in part through use of the object provided by a fourth process (D). When A needs to use the object it knows about, it enters a dialog (ALPHA) with B, as in Figure 8.6b. In order to perform the required manipulation of the object, B must enter a dialog (BETA) with C. Since BETA occurs during ALPHA, C must also enter ALPHA before entering BETA with B. At this point, as Figure 8.6c shows, A is in (to use the internal names) dialog ALPHA, and B and C are in dialog ALPHA.BETA within dialog ALPHA. In order to perform the manipulation required of it, C must communicate with D in another dialog (GAMMA). At this point, we have the situation of Figure 8.6d, that A is in dialog ALPHA, B is in dialog ALPHA.BETA within dialog ALPHA, and C and D are in dialog ALPHA.BETA.GAMMA within dialog ALPHA.BETA within dialog ALPHA.

One naturally wonders how D knew what dialog names to use, and how deeply it had to go into dialog nesting to render its service. We do not have an answer for that question. Indeed, we view the acquisition of that kind of knowledge as completely counter to modern design and programming precepts such as modularity, information hiding and abstraction. The planned establishment of recovery lines interferes with modularity and information hiding.

8.3. Some Naive Approaches to Solutions

There are several approaches that might occur to one seeking solutions to these problems. Many of these are dead-ends. We have already mentioned and rejected having service processes cycle among clients' potential dialogs and exiting these using TIMEOUTS. We present a few more approaches here with explanations of why they do not solve the problem.



Dialog Nesting Counter to Modularity

Figure 8.6

Conditional Dialogs

A service process is a special kind of process. Many others try to communicate with it, but the others don't care at all about each other. A service process's design wants to be a loop with a selective wait in it.

An approximation of nondeterminism can be reintroduced outside the dialog envelope by the programmer. The robot example in Appendix C approximates a nondeterministic choice among callers. The nondeterminism had to be faked in the robot example — a complex contrivance was used which should not be left to the programmer.

Ada allows a task to make nondeterministic choices among entries when accepting calls. There is no corresponding nondeterminism when choosing to enter a dialog. In a `dialog_sequence`, the keyword OR precedes a *sequential* alternative. However, in the Ada selective wait, the OR means a *nondeterministic* "choice" is to be made.

One possible way for a process to choose among dialogs nondeterministically would be to revise the language to replace the "accept part" in a selective wait with a `dialog_sequence`. We call this approach the "conditional dialog". Of course, implementing a nondeterministic entry into one of several dialogs would be extremely difficult. Furthermore, such an approach does not necessarily achieve the necessary dialog nesting depth.

Limited Participation Dialogs

Ada also enforces mutual exclusion among entry calls being serviced. The dialog allows any process to enter the communication at will.

To deal with the mutual exclusion problems of all participants failing undeservedly or all succeeding mistakenly, we might consider using Path expressions [Camp74, Camp80] or something like them for the number of processes in a dialog.

A form of this is participant counts. If a `DISCUSS` statement has only accepts within its sequence of statements, it may be reasonable for the service process to create a sub-dialog for a specified number of participants by specifying a participant count. Then, only as many processes on the entry queue as were specified are considered actually to be "in" the dialog with the acceptor. Again, this idea complicates the bookkeeping necessary for implementing dialogs.

8.4. Possible Research Directions Toward Real Solutions

There is reason to argue that backward recovery should be applied to larger sets of communications between processes than individual rendezvous. We agree, but the nature of service processes is such that they only have short communications with clients or must effectively time-share themselves among clients. The whole scheme of `dialog_sequence`s seems inappropriate for microscopic communication as defined by service processes and similar practices. We are thus tempted to seek more radical approaches to recovery after communication.

Below, we outline our current thoughts on two possible directions for future research toward an actual solution. They are not intended to be complete, nor do they necessarily address all of the points made in the preceding paragraphs.

8.4.1. Limited Recoverable Objects

An initial approach to this problem would be to require program structures that do not use service processes. This is a severe, perhaps unrealistic restriction given current design techniques for concurrent programs. A second approach might be to designate service processes as somehow special cases, and allow them to operate outside of the dialog rules and arrange their own recovery.

The combination of backward error recovery with independence of process designs for protecting shared variables and external hardware suggests a new facility not found in most languages for this applications area. This facility is a *limited recoverable object*. It is an abstraction with which processes communicate but which decides independently when to recover or to commit to a new state.

Changes to values of limited-recoverable objects would be recoverable only for a short span of "time", after which their values would become unrecoverable. Such objects may well deserve a special syntax and semantics for dealing with them. With a specialized syntax and semantics, limited recoverable objects might also rid us of the problem of the service process needing to know its clients and their dialog nesting structures.

The operating systems examples' problems [Well84] (such as producer/consumer) with backward recovery is what Russell [Russ77] was talking about when he said to unbundle the caching operations SAVE, RESTORE, and DISCARD. Unfortunately, his unbundling of the operations only opens up more room for faults to appear. It may be possible systematically to embed the concept of a limited recoverable object in the language so as to prevent many faults of misuse of the operations. One process would communicate with such an object at a time and they would either commit or recover before disengaging. Alternatively, the service process implementing the limited recoverable object might keep track of recovery points for each client's transactions with it, and independently determine when to recover or commit to that transaction. The decision to commit or recover would be almost entirely the responsibility of the object.

Limited recoverable objects would also be useful in modeling a program's communication with the outside world. Such an object could serve as a compromise between the unrecoverable objects of the external world and the remainder of the program.

Without the exercise of due care in developing this approach, it could be abused easily and could substantially reduce the compile-time checking of the use of backward error recovery that is possible with the general dialog scheme.

8.4.2. Multi-Level Solution

The levels of abstraction and of service processes are reminiscent of Anderson's multi-level programs [Ande78]. As a layer of a multi-level system, each service process might be made independently responsible for recovering itself. We might extend the ideas of limited recoverable objects to other units of the programming language. Of course, inactive units would have to be prompted by the support system to perform recovery. For instance, there could be two levels of specifications for shared packages, one seen by the user and another seen by the recovery system[†]. The recovery system could call the audit code periodically to evaluate the package's private variable state and could call the recovery code when needed. Part of the audit code could employ a kind of message validation on incoming and outgoing parameter lists.

If recovery is triggered by one client, not all clients need be backed up. An audit trail could be used to replay old ''messages'' from other clients, thus using an actual previous state of that layer to construct a state which could have existed. As the system re-played messages from other clients since the last recovery point of the layer that was recovering, checks could be made that outgoing parameters to other clients matched those in actual old messages. Unfortunately, a problem could arise if the new and old outgoing parameters did not match. To have the server send out ''corrective'' messages to other clients would not be desirable, especially since the server (as opposed to the recovery system) is not supposed to know who those clients are.

[†] The visible and private parts of package specifications in Ada already represent multiple levels for user and compiler, respectively.

It may be that the only alternatives are to employ a chase protocol [Merl78] to trigger recovery in other clients, or to have the server ''stonewall it''. In the former case, the domino effect enters the picture once more. In the later case, the server would recover on its own and let its clients detect their new inconsistency with the protected data of their own accord. Either case is very dangerous.

Some of the ideas above may be recognizable as having been employed in the DMERT system [Hans83]. Before being employed within programming language facilities for backward error recovery, any ideas must be organized and integrated with the other facilities of the language so that mechanisms for subverting the backward error recovery facilities can be detected long before execution-time.

8.5. Summary

In this chapter, we have expanded upon the problem of incompatibility between the planned establishment of recovery lines and use of modern programming practices and design methods. We concentrated on service processes and one-way naming which facilitate information hiding, abstraction, and object-oriented design. We identified several effects and consequences of blindly including backward error recovery in a system with service processes. While illustrating these effects, we included and simultaneously showed the inefficacy of several naive solutions to the incompatibility problem. We then showed a few other dead-end approaches to the problem.

In previous chapters, we proceeded to present real solutions to the problems. In this chapter, we have delved instead slightly deeper into the cause(s) of the incompatibility problem, and have outlined some approaches that may prove fruitful in future investigations. We have not found a satisfactory solution to the problem that produces the capture effect, and emphasize that it is but one example of the general issue of program structure for systems incorporating backward error recovery.

Our focus has been on the identification of a heretofore unrecognized problem. Our search for a solution has reinforced the conclusion that backward error recovery must be a design consideration in a programming language rather than added as an afterthought.

CHAPTER 9

Summary, Conclusions, and Future Work

9.1. Summary

In this dissertation, we have been concerned with programming language primitives that allow the specification of fault-tolerant concurrent systems, and the impact that such primitives have on other elements of modern programming languages.

In Chapter 2, we described the previous major proposals for providing software tolerance of software faults. These are the proposals that have included or have hinted at the provision of programming language support rather than merely being design recommendations. These included both forward and backward error recovery. We continued to delineate many shortcomings of the backward error recovery proposals.

Chapter 3 proposed a solution to the problems that we had pointed out in the previous backward error recovery proposals. The solution is a new backward error recovery primitive and an organization for its use. We named these the dialog and colloquy, respectively. We also defined programming language facilities that allow the specification of dialogs and of processes' participation in colloquys. These, we named the `DISCUSS` statement and `dialog_sequence`, respectively.

Appendix A gives detailed semantics of the `DISCUSS` statement and `dialog_sequence`, and gives a possible syntax that would allow static enforcement of their proper use.

Chapter 4 demonstrated by high-level description that the `DISCUSS` statement and `dialog_sequence` can be implemented in a programming language.

Appendix B demonstrates that the colloquy is a more general concept than the previous backward error recovery proposals. It does this by translating each of the previous proposals into a special case of `dialog_sequence`S.

By examining the inclusion of the new backward error recovery facilities in a realistic modern programming language, we revealed problems that are more fundamental than the previous proposals' shortcomings but that had been obscured by them or that had been simply ignored or ''assumed away'' in the past. The problems evidence themselves in the form of smuggling and of incompatibilities in the explicit communications philosophies of backward error recovery and the language.

These new problems were introduced in Chapter 5 to be addressed more closely in Chapter 6, Chapter 7, and Chapter 8. In those chapters, the problems were illustrated using examples of execution-time situations that might arise if nothing (or the wrong thing) were done about the problems. The illustrating situations are not at all pathological cases. Rather, they are very simplified statements of the complex situations that can arise in actual programs.

In Chapter 6, we addressed the problem of smuggling via programming language facilities that one would expect could be used to that end. We elaborated on this problem both through exposition and by presenting naive approaches to solution along with their shortcomings. To demonstrate the problem, we examined shared variables, aliasing, and ''pointers''. We call our solution to the problems of smuggling via shared variables and aliasing, IMAGEs. Through explicit declaration and constrained usage of any sharable variable, IMAGEs provides an elegant anti-smuggling wall around the activities of processes participating in a dialog. We next showed that constraints designed to prevent smuggling through pointers, or dynamically allocated objects, would severely limit pointers' usefulness.

In Chapter 7, we addressed the problem of smuggling via programming language facilities that one would not expect could be used to that end. One unexpected source of information flow is the manipulation of processes. We examined three mechanisms for process manipulation and showed how they permit smuggling when backward error recovery is naively combined with them. We used as examples dynamic process creation, dynamic process destruction, and examination of processes' execution states. We showed that there are several approaches that one might consider in seeking a solution which are unworkable. Our solution to smuggling due to creation of dynamically allocated processes prescribes dividing the body of a process into a preliminary part and a main part and providing a FINAL function describing the process' chance to veto its own creation. We also showed how smuggling can arise due to the combination of abnormal destruction of processes (ABORT, in Ada) and examination of processes' execution states (task attributes, in Ada).

Finally, in Chapter 8, we expanded upon the problem of incompatibility between the needs of backward error recovery and program structures governed by the communications philosophy of the language. That philosophy reflects the precepts of modern programming practices and design methods. Those appropriate to Ada require program organizations that run counter to the needs of processes in the establishment of planned recovery lines. We delved into the causes of the incompatibility problem, and outlined some solution approaches that may prove fruitful in future investigations.

9.2. Conclusions

We have defined a primitive, the dialog, that is a basic unit from which backward error recovery schemes can be constructed. We have introduced a new linguistic construct, the colloquy, which solves the problems identified in the earlier proposal, the conversation. The colloquy generalizes all previously proposed linguistic constructs for backward error recovery.

The major features that distinguish the colloquy over the conversation are:

(1) The reversal of the order of priority of alternate communication attempts and of recovery points. This allows processes to choose the participants in any alternate algorithms rather than being required to deal with a single set of processes.

(2) The inclusion of explicit and general timing constraints. This allows processes to protect themselves against any difficulties in communication that might prevent them from meeting real-time deadlines. It also deals with the problem of deserter processes.

(3) The use of a two-level acceptance test. This allows much more powerful error detection because it allows the tailoring of acceptance tests to specific needs.

Subsequent to defining the colloquy, we attempted to introduce the concept into a *realistic* programming language. That is, a programming language that contains the necessary facilities to program modern, large-scale applications. It is tempting to think that the colloquy solves all the problems that might arise, and that the syntax for the colloquy can be integrated into a programming language with no further concern. When this was attempted, problems that are at least as serious as the previous ones were revealed. The difficulties arise because of an inherent *conflict* between the needs of the applications' programmer, as exemplified by modern language design, and the fundamental needs of backward error recovery in concurrent systems. These problems fall into the general categories of *smuggling* and *program structure*.

All of the newly-uncovered difficulties that we discussed use Ada as an example. The actual programming language is irrelevant, and we stress the fact that the problems raised *will* occur in other programming languages. Nor are these problems specific to the colloquy. They arise because of the fundamental requirements of backward error recovery in concurrent systems, and occur with the conversation also. In general, other researchers have either assumed these problems would not occur, or based their research on programming languages with very limited (and impractical) facilities, such as CSP [Jalo84]. The problems do not arise in those languages because of the relatively simple semantics that such languages have.

We have shown how some of these difficulties can be overcome by making extensive changes to the programming language. All of our proposed solutions were developed for a particular class of programming languages exemplified by Ada. The proposed solutions are unlikely to handle variations in language design. Neither our approaches to pointers nor our solution to shared variables would solve the problems with, for example, PL/I's based variables. This shows how awkward the semantics of existing languages can be.

We have dealt with only a few of the expected mechanisms for smuggling, and only a few of the unexpected mechanisms for smuggling. There may be others, if not in Ada, then in other languages. The incompatibility problem is also but one example of the general issue of program structure for systems incorporating backward error recovery.

Backward error recovery and many modern programming language facilities in combination present serious problems not previously addressed or even acknowledged in the literature. Our focus has been on the identification of heretofore unrecognized problems. Our search for solutions to the smuggling and incompatibility problems has been in the form of alterations to the semantics of the `dialog_sequence` and minor alterations to the semantics of the language. We have seen many apparent solutions which, upon closer examination, have proven unsatisfactory.

All of the problems described derive from the rapid growth in size and complexity of programming languages with no attention to backward error recovery. As a result, the asynchronous nature of processes makes it difficult to establish a recovery line and certainly to pre-package it with a syntactically enforceable structure in the manner that a `DISCUSS` statement is enforceable for explicit messages.

The concepts of backward error recovery cannot be merged *naively* into existing realistic programming languages. Our search for solutions has reinforced the conclusion that backward error recovery must be a design consideration in a programming language rather than added as an afterthought. It is clear that backward error recovery cannot be included in a programming language after the fact. It must be a design consideration from the outset. The final solution has to be the design of an entirely new programming language with backward error recovery as its starting point.

9.3. Future Work

The obvious course for future work, given our conclusions, is to attempt to design a language with backward error recovery as a major driving force. A test for success of such a project would be whether the resulting language retained forms of the other facilities that would make it useful in programming embedded systems. Below are some of the considerations that the language designer should take into account.

In the non-backward-error-recovery view, only data and processes exist at execution-time. Consideration of backward error recovery adds recovery lines (dialogs) to this list. As one of the three kinds of entities at execution-time, recovery lines should be a major driving factor in processes' activities. Processes' activities are specified at compile time. So, any compile time specification needs to obey the dialog rules. The behaviors known as "service processes" conflict with planned establishment of recovery lines. Hence, a structure other than service processes is required.

At execution time, data comes and goes, and we recognize that the same datum can be reached via different source identifiers at different times and may even be unreachable at times via any identifier. We have the same recognition about processes and the identifiers used in reaching them. *Now* we need to deal with execution-time-only entities variously called dialogs or recovery lines or coordinated sets. For both of the previously recognized classes of execution-time objects, we know that sometimes an identifier will reach the entity, but we are better off (under rules or restrictions) not using it, or using it under a protocol. This will also be true of dialogs.

The DISCUSS statement as we have defined it provides for the general case in which all participants desire a recovery wall around their communication. We might consider modified DISCUSS statements for specialized usage of dialogs. We currently refer to these as local and non-local dialogs. One case is that of a single process which is concerned about its own recovery but not communicating. Another case is a small group of processes concerned about their own recovery yet which need to communicate with outside servers within that time whether the servers care to be able to recover or not. That case would provide for explicit use of limited recoverable objects.

It may be the case that languages suitable for backward error recovery and languages suitable for verification are basically the same. This might also instigate a radical change in the form of dialogs or of processes.

Beyond the design of a language with backward error recovery as a major driving force, and considering the trends toward distribution of embedded systems, investigation of the resulting language's efficient implementability using distributed algorithms would be a reasonable further continuation.

APPENDIX A

Syntax and Detailed Semantics of the Dialog Sequence

The Syntactic entities `dialog_sequence`, `discuss_statement` and `fail_statement` facilitating use of the dialog and colloquy in an Ada-successor language are detailed here.

Figure A.1 shows the syntax associated with the colloquy. A `discuss_statement` contains both a global test designation and a local test. The global test designation is a name that has been declared in a `dialog_declaration`. A `dialog_declaration` associates the name of a parameterless boolean function with a list of names of variables used within the function and within any `discuss_statement` which uses the name of that function as its `dialog_name`. The local test is simply a boolean expression.

The keyword ''NEVER'' in the TIMEOUT part indicates that the programmer places no timing requirement on the list of attempts in that `dialog_sequence`. In the absence of a timing requirement, the syntax chosen for a `dialog_sequence` forces the programmer to say so explicitly. The syntax given for the TIMEOUT part, ''TIMEOUT'', may be disturbing to some who may prefer something like ''OR DELAY'', in which case the keyword ''NEVER'' may be changed to ''FOREVER''.

Since it represents a time interval, in Ada the simple expression following the ''TIMEOUT'' keyword would be of type DURATION. Should the simple expression return a negative number or zero, control is transferred immediately to the sequence of statements in the TIMEOUT part.

The sequence of statements in the TIMEOUT part may be empty. Execution of an absent sequence of statements, or a sequence consisting only of the NULL statement causes the `dialog_sequence` and whatever surrounds it to become unsuccessful. The sequence of statements in the TIMEOUT part allows the programming of a ''last ditch'' algorithm for the task to achieve its goal in the case in which a timing

```
select_statement      ::= selective_wait
                      |   conditional_entry_call
                      |   timed_entry_call
                      |   dialog_sequence

dialog_sequence       ::= SELECT
                              attempt_list
                              timeout_part
                              el,e_part
                          END SELECT;

attempt_list          ::=      attempt
                          { OR attempt }

attempt               ::=   discuss_statement
                          [ sequence_of_statements ]

discuss_statement     ::= DISCUSS dialog_name BY
                             sequence_of_statements
                          TO ARRANGE boolean_expression;

dialog_declaration    ::= DIALOG function_name
                             SHARES ( name_list );

timeout_part          ::= TIMEOUT NEVER;

                      |   TIMEOUT simple_expression;
                            [ sequence_of_statements ]

else_part             ::= ELSE
                            [ sequence_of_statements ]

statement             ::= .... | FAIL;

-- A task participates in a dialog by executing a discuss_statement.
-- A discussion is determined by a task executing a dialog_sequence.
```

Syntax for the Colloquy

Figure A.1

constraint has been violated during processing of the attempts of this `dialog_sequence`. If this sequence of statements is successful, control continues after the `dialog_sequence`. If the `TIMEOUT` part is not successful, further actions are as if the `TIMEOUT` part had been empty.

A sequence of statements is permitted following the `discuss_statement` within an attempt. This sequence of statements is logically outside of the encompassing `dialog_sequence`. As such, if the associated `discuss_statement` is successful, it is executed after the recovery point is discarded but before execution of any statement which textually follows the `dialog_sequence`. It is intended for specialized post processing of the activities within its associated `discuss_statement`. Since it is logically outside of the enclosing `dialog_sequence`, it is not subject to any timing constraint which may have been specified in that `dialog_sequence`'s TIMEOUT part. Also since it is logically outside of the enclosing `dialog_sequence`, if this sequence of statements becomes unsuccessful, it is not the attempt but the sequence of statements containing the `dialog_sequence` which becomes unsuccessful.

If control is transferred to an ELSE part whose sequence of statements is absent, or if the sequence consists of a single NULL statement, the `dialog_sequence` and whatever surrounds it becomes unsuccessful. Figure A.2 shows three ELSE parts with equivalent semantics. The sequence of statements in the ELSE part affords the programmer the opportunity to specify some ''last ditch'' actions for the task to achieve its goal. If this sequence of statements is successful, control continues after the `dialog_sequence`. If the ELSE part is not successful, further actions are as if the ELSE part had been empty. Like the TIMEOUT part, the ELSE part does not protect the sequences of statements following

```
 ELSE
END SELECT;

 ELSE
      NULL;
END SELECT;

 ELSE
      FAIL;
END SELECT;
```

Three Equivalent ELSE Parts

Figure A.2

`discuss_statements` within attempts.

A `dialog_sequence` is a full statement and need not be restricted to a task body if it contains no task communication statements. There is no requirement that an attempt actually contain communications statements. Thus, the construct can be used to build a recovery block for a sequential part of a program.

`selective_wait`s, `conditional_entry_call`s, `timed_entry_call`s, `entry_call_statement`s, and `accept_statement`s may only occur within `discuss_statements`. No communications statements are allowed in the `ELSE` part or the `TIMEOUT` part or the sequences of statements following `discuss_statements` within attempts, at least for the task's outermost `dialog_sequence`s.

Tasks have a new attribute (see Figure A.3) associated with dialogs. This attribute returns a Boolean value signifying whether the task `T` is, at the time the attribute is evaluated, a participant in the implied dialog. In the case of nested dialogs, the innermost is meant. The task evaluating the attribute (requesting the information) must itself be in some dialog. The main intended use for this attribute is in the global acceptance test. There, the test can be made to fail due to desertion by certain ''ex-officio'' members of the dialog. A more general form (see Figure A.4) would be too difficult to use in the global tests' bodies. The body would have to be tailored for a particular level of nesting of dialogs to provide the appropriate dotted dialog name. Another reason for restricting the attribute is to prevent the interrogator from smuggling out meta information about dialogs in which it is not a participant.

```
T'in_this_dialog
```

The `in_this_dialog` Attribute

Figure A.3

```
T'in_dialog( <dialog_name> )
```

A More General Attribute

Figure A.4

APPENDIX B

Other Software Fault-Tolerance Facilities as Special Cases

dialog_sequences can be used to construct deadlines, generalized exception handlers [Salz78], recovery blocks, traditional conversations, exchanges, and s-conversations [Jalo84]. Thus the colloquy is at least as powerful as each of these previously proposed constructs for provision of fault tolerance.

The Deadline Mechanism

Figure B.1 shows Campbell's deadline mechanism implemented with a dialog_sequence. The order of primary and alternate actions in a dialog_sequence basically requires the use of Campbell's "optimistic" scheduling scheme in which the primary is given the chance to execute first and the alternate starts when only enough time remains for it to operate. This example shows the construction of exactly what Campbell describes, with the same problems. The time given is that allowed the primary rather than, as Campbell prescribes, the timing requirement for the entire statement. Campbell assumes that the time required by the alternate is known. If the timing requirement for the entire statement and for

```
SELECT
  attempt -- read sensors; calculate new position;
 TIMEOUT 7 * millisecond; -- deadline duration minus time of alternate.
  stmts; -- approximate new position from old position.
 ELSE
  stmts; -- same backup algorithm as above.
END SELECT;
```

Colloquy for Campbell's Deadline Mechanism

Figure B.1

the alternate algorithm are known, it should be possible to determine that for the primary algorithm. Also notice that the ELSE and TIMEOUT parts are identical. Campbell's deadline mechanism does not distinguish between timing errors and other faults in the primary algorithm. He even assumes the alternate algorithm is correct, so we have not protected the algorithm in the ELSE and TIMEOUT parts.

A Generalized Exception Handler

Figure B.2 shows the use of a dialog_sequence to construct a general exception handler. Most languages with exceptions do not deal with timing, which is the only discrimination the dialog_sequence makes; hence, we have not given a timing constraint in Figure B.2. FAIL statements can provide an approximation of Ada "raise" statements in either of the statement sequences in Figure B.2. This is still more powerful than Ada's handler for OTHERS (Figure B.3) since, in the dialog_sequence, all changes made during the failed attempt are undone before execution of the ELSE part begins. In that sense, the example more resembles Cristian's ideas on exceptions than Ada's. Although no particular reason is made apparent for why things went wrong in the attempt, a consistent

```
FUNCTION abc RETURNS boolean IS BEGIN RETURN TRUE; END abc;
  ....
DIALOG abc SHARES ( );
  ....
SELECT
  DISCUSS abc BY
    sequence_of_statements_1 -- this is the block being protected
  TO ARRANGE TRUE;
 TIMEOUT NEVER;
 ELSE
  sequence_of_statements_2 -- this is the handler
END SELECT;
```

Colloquy for Exception Handler for OTHERS

Figure B.2

```
... BEGIN
       sequence_of_statements_1 -- this is the block being protected
    EXCEPTION
       WHEN OTHERS =>
          sequence_of_statements_2 -- this is the handler
    END ...
```

Ada Exception Handler for OTHERS

Figure B.3

state has been established from which the handler can operate in its job of replacing the protected code.

The Recovery Block

A recovery block is a special case of a colloquy in which there is only one process participating, every dialog uses the same acceptance test, there is no timing requirement, and there are no "last ditch" algorithms to prevent propagation of failures of the construct. Figure B.4 shows a `dialog_sequence` that is equivalent to the recovery block shown in Figure B.5. The use of the `FAIL` statement in the `dialog_sequence` makes explicit the propagation of the error to a surrounding context just as does the `else error` closing of the recovery block. In the `dialog_sequence`, the Boolean expression is repeated in the `DISCUSS` statements rather than gathered into the dialog function because we want to be able to include local variables in it as a programmer of the recovery block would. Should an error be detected in `statement_sequence_1`, the state is restored and `statement_sequence_2` is executed, and so on. Finally, should an error be detected in `statement_sequence_3`, the state is restored and the error is signaled in a surrounding context. An error may be detected by evaluation of `boolean_expression_1` to `FALSE`, or by violation of some underlying interface (such as raising of an exception).

```
FUNCTION abc RETURNS boolean IS BEGIN RETURN TRUE; END abc;
  ....
DIALOG abc SHARES ( );
  ....

SELECT
      DISCUSS abc BY
         statement_sequence_1
      TO ARRANGE boolean_expression_1;

   OR
      DISCUSS abc BY
         statement_sequence_2
      TO ARRANGE boolean_expression_1;

   OR
      DISCUSS abc BY
         statement_sequence_3
      TO ARRANGE boolean_expression_1;

   TIMEOUT NEVER;

   ELSE
       FAIL; -- Omitting this line does not change the semantics.
END SELECT;
```

Specification of Colloquy for a Recovery Block

Figure B.4

Traditional Conversations

Since a `dialog_sequence` can be used to construct a recovery block, and since it can contain communications statements, a set of `dialog_sequences` within a collection of tasks can also be used to construct a traditional conversation. To do this, the `ELSE` and `TIMEOUT` parts must be empty. The `TIMEOUT` part is empty since a traditional conversation cannot detect desertion, which is a timing consideration. The `dialog_sequence` in each task must have the same number of attempts as the others. All attempts must have the same dialog name. The dialog function must simply return `TRUE`, and no shared variables should be named in the dialog declaration. Finally, the boolean expressions in all attempts of any one `DISCUSS` statement must be identical and represent that task's contribution to the

```
ENSURE boolean_expression_1 BY
   statement_sequence_1

ELSE BY
   statement_sequence_2

ELSE BY
   statement_sequence_3

ELSE ERROR;
```

A Recovery Block

Figure B.5

acceptance test for the conversation. To prevent smuggling, besides admitting no shared variables, no other task in the system must be allowed to use the `dialog_name` used by these tasks for this conversation.

Exchanges

Exchanges can also be constructed using `dialog_sequences` as is illustrated in Figure B.6. Each task is an infinite loop to read sensor values, process them, re-synchronize with the other tasks, and output the results. One pass through the loop represents one frame. Should the processing take too long or otherwise fail, some emergency actions can be available such as leaving the old (now restored) values in the latch variables and noting how often this has occurred.

The comment within the `DISCUSS` statement suggests an alternate design in which only a parent task would contain the loop. During its `DISCUSS` statement it would create the tasks that were to do the actual work, delay for some large fraction of the available frame time while they worked, then destroy them or, if they had not finished by the time it re-awakened, execute a `FAIL` statement. This latter design has much task creation overhead but more directly models exchanges.

```
FUNCTION abc RETURNS boolean IS BEGIN
   -- check up on the latch variables and inputs, etc.
END abc;
  ....
DIALOG abc SHARES (
                   -- list of inputs and latch variables
                  );
....
LOOP -- in each task:
   -- read sensors, etc.
   SELECT
    DISCUSS abc BY
       sequence_of_statements
       -- (could even start up the tasks and kill them)
    TO ARRANGE test;
    TIMEOUT cycle_length;
       -- some statements to absorb the failure
    ELSE
       -- some statements to absorb the failure
   END SELECT;
   DELAY whatever_it_takes_to_not_go_too_fast;
   -- output contents of latch variables to actuators
END LOOP;
```

Colloquy for Exchanges

Figure B.6

S-conversations

S-conversations are conversations with synchronized entry and explicitly-named participants. The concept was introduced in the context of CSP which has named send and named receive in its communication statements. In that context, the programmer might as well name all participants in one place. If each process goes into a loop at the beginning of its attempt waiting on the condition that all other participants have also arrived in the dialog, the initial synchrony is achieved easily. Thus, we have a use outside of the acceptance test for the new `in_this_dialog` attribute which we have associated with dialogs.

APPENDIX C

Controller for a Robot Arm

This appendix illustrates some uses of dialogs in a set of tasks implementing a controller for a robot arm. The tasks responsible for the coordinated delicate operations of the ''fingers'' and ''wrist'' operate at the minor frame rate. Those responsible for the gross movements of ''elbow'' and ''shoulder'' operate at the major frame rate. Coordination of the two major parts is through a shared positioning database guarded by a service task. For simplicity, we ignore such things as acquisition of orders (that would probably be communicated through the database). At the end of its frame, a representative of each part exchanges information with this database.

In the ''hand'', each participant reads its sensors, then makes (potentially) two attempts to coordinate its decision about signals to its actuators with its partner in light of the most recently available information about what the upper arm is doing. At the end of its frame, the participant sends out its control signal. Should the attempts take too long or both fail, the previous frame's output or an extrapolation from it is used.

The wrist, representing the hand, communicates the group's decisions to the database, at the same time retrieving pertinent information about recent upper arm activities. Should that take too long, or should some disagreement with the database arise, temporary assumptions are made about the arm's status.

The upper arm operates similarly to the hand.

The task guarding the database cycles through dialogs employing selective waits with ELSE part. Thus, when a representative of one of the groups calls, the task is situated within the correct dialog. The

RENDEZVOUS variable insures that the dialog will not fail for lack of a caller. If there is a caller and the dialog fails, the database is restored to a consistent state automatically and the failure is not propagated.

Words containing "_GOAL_IN_" stand for task- and dialog- specific local acceptance tests. As such, they are application-specific and probably quite complicated. Thus, we have abbreviated them.

The text of the robot arm controller follows:

```
task FINGERS is
   -- entries for FINGERS
end FINGERS;

task WRIST is
   -- entries for WRIST
end WRIST;

task POSITION is
   entry HAND( ... );
   entry ARM( ... );
end POSITION;

task ELBOW is
   -- entries for ELBOW
end ELBOW;

task SHOULDER is
   -- entries for SHOULDER
end SHOULDER;

dialog DELICATE_WORK is
begin
   -- evaluate primary global goal of FINGERS and WRIST returning a Boolean.
end DELICATE_WORK;

dialog ALT_DELICATE is
begin
   -- evaluate alternate global goal of FINGERS and WRIST returning a Boolean.
end ALT_DELICATE;

dialog GROSS_WORK is
begin
   -- evaluate primary global goal of ELBOW and SHOULDER returning a Boolean.
end GROSS_WORK;

dialog ALT_GROSS is
begin
   -- evaluate alternate global goal of ELBOW and SHOULDER returning a Boolean.
end ALT_GROSS;

dialog HAND_LOCATION is
begin
   -- evaluate global goal of WRIST and POSITION returning a Boolean.
end HAND_LOCATION;

dialog ARM_LOCATION is
begin
   -- evaluate global goal of ELBOW and POSITION returning a Boolean.
end ARM_LOCATION;

task body FINGERS is
```

```
begin
   loop

      -- pick up inputs (unrecoverable).

      select
         discuss DELICATE_WORK by
            -- many rendezvous with WRIST while computing new outputs.
         to arrange FINGERS_GOAL_IN_DELICATE_WORK;
      or
         discuss ALT_DELICATE by
            -- many rendezvous with WRIST using alternate algorithm
         to arrange FINGERS_GOAL_IN_ALT_DELICATE;
      timeout PART_OF_MINOR_CYCLE;
         -- emergency: use old outputs.
      else
         -- emergency: extrapolate from old outputs.
      end select;

      delay TO_MINOR_FRAME_BOUNDARY;

      -- send outputs (unrecoverable).

   end loop;
end FINGERS;

task body WRIST is
begin
   loop

      -- pick up inputs (unrecoverable).

      select
         discuss DELICATE_WORK by
            -- many rendezvous with FINGERS while computing new outputs.
         to arrange WRIST_GOAL_IN_DELICATE_WORK;
      or
         discuss ALT_DELICATE by
            -- many rendezvous with FINGERS using alternate algorithm
         to arrange WRIST_GOAL_IN_ALT_DELICATE;
      timeout PART_OF_MINOR_CYCLE;
         -- emergency: use old outputs.
      else
         -- emergency: extrapolate from old outputs.
      end select;

      select
         discuss HAND_LOCATION by
            POSITION.HAND( ... );
         to arrange WRIST_GOAL_IN_HAND_LOCATION;
      timeout ANOTHER_PART_OF_MINOR_CYCLE;
         -- emergency: use old arm info.
      else
         -- emergency: extrapolate from old arm info.
      end select;

      delay TO_MINOR_FRAME_BOUNDARY;

      -- send outputs (unrecoverable).

   end loop;
end WRIST;

task body POSITION is
   -- declarations for the (protected) shared information go here.
   RENDEZVOUS : boolean;
begin
   loop

      select
         discuss HAND_LOCATION by
            RENDEZVOUS := true;
            select
               accept HAND( ... ) do
                  -- store hand position; communicate arm position
               end HAND;
            else
               RENDEZVOUS := false;
            end select;
         to arrange CONSISTENT_POSITION_FOR_ROBOT or not RENDEZVOUS;
      timeout never; -- this service task depends upon clients to time calls.
      else
         RENDEZVOUS := false; -- meaningless statement,
                              -- just stops failure propagation.
      end select;

      select
         discuss ARM_LOCATION by
            RENDEZVOUS := true;
            select
               accept ARM( ... ) do
                  -- store arm position; communicate hand position
               end ARM;
            else
               RENDEZVOUS := false;
            end select;
         to arrange CONSISTENT_POSITION_FOR_ROBOT or not RENDEZVOUS;
      timeout never; -- this service task depends upon clients to time calls.
      else
         RENDEZVOUS := false; -- meaningless statement,
                              -- just stops failure propagation.
      end select;

   end loop;
end POSITION;

task body ELBOW is
begin
```

```
   loop

      -- pick up inputs (unrecoverable).

      select
         discuss GROSS_WORK by
            -- many rendezvous with SHOULDER while computing new outputs.
         to arrange ELBOW_GOAL_IN_GROSS_WORK;
      or
         discuss ALT_GROSS by
            -- many rendezvous with SHOULDER using alternate algorithm
         to arrange ELBOW_GOAL_IN_ALT_GROSS;
      timeout PART_OF_MAJOR_CYCLE;
         -- emergency: use old output.
      else
         -- emergency: extrapolate from old outputs.
      end select;

      select
         discuss ARM_LOCATION by
            POSITION.ARM( ... );
         to arrange ELBOW_GOAL_IN_ARM_LOCATION;
      timeout ANOTHER_PART_OF_MAJOR_CYCLE;
         -- emergency: use old hand info.
      else
         -- emergency: extrapolate from old hand info.
      end select;

      delay TO_MAJOR_FRAME_BOUNDARY;

      -- send outputs (unrecoverable).

   end loop;
end ELBOW;

task body SHOULDER is
begin
   loop

      -- pick up inputs (unrecoverable).

      select
         discuss GROSS_WORK by
            -- many rendezvous with ELBOW while computing new outputs.
         to arrange SHOULDER_GOAL_IN_GROSS_WORK;
      or
         discuss ALT_GROSS by
            -- many rendezvous with ELBOW using alternate algorithm
         to arrange SHOULDER_GOAL_IN_ALT_GROSS;
      timeout PART_OF_MAJOR_CYCLE;
         -- emergency: use old outputs.
      else
         -- emergency: extrapolate from old outputs.
      end select;
      delay TO_MAJOR_FRAME_BOUNDARY;

      -- send outputs (unrecoverable).

   end loop;
end SHOULDER;
```

References

Ada83.

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

Ambe77.

A. L. Amber, D. I. Good, J. C. Browne, W. F. Burger, R. M. Cohen, C. G. Hoch, and R. E. Wells, "Gypsy: A Language for Specification and Implementation of Verifiable Programs," *SIGPLAN Notices*, **12**(3), pp. 1-10 (March 1977).

Ande78.

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).

Ande81.

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

Ande83a.

J. L. Anderson, "Space Station Autonomy Requirements," pp. 164-170, in *AIAA Computers in Aerospace IV Conference*, American Association of Aeronautics and Astronautics, Hartford, Conn. (October 1983).

Ande83b.

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).

Aviz77.

A. Avizienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault-Tolerance During Program Execution," *Proceedings COMPSAC 77*, pp. 149-155 (November 1977).

Bens84.

D. B. Benson, "Letter on Disaster Details," *ACM Software Engineering Notes*, **9**(1), pp. 10-11 (January 1984).

Bern78.

P. A. Bernstein, J. B. Rothnie, N. Goodman, and C. A. Papadimitriou, "The Concurrency Control Mechanisms of SSD-1: A System for Distributed Databases (The Fully Redundant Case)," *IEEE Transactions on Software Engineering*, **4**(3), pp. 154-168 (May 1978).

Bern81.

P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, **13**(2), pp. 185-222 (June 1981).

Best80.

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

Blac81.

J. P. Black, D. J. Taylor, and D. E. Morgan, "A Case Study in Fault Tolerant Software," *Software Practice and Experience*, **11**(2), pp. 145-157 (February 1981).

Bohe73.

B. W. Bohem, "Software and its Impact: A Quantitative Assessment," *Datamation*, **19**(5), pp. 48-59 (May 1973).

Brin77.

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

Camp74.

R. H. Campbell and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," *Lecture Notes in Computer Science* (1974).

Camp79.

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).

Camp80.

R. H. Campbell and R. B. Kolstad, "An Overview of PATH PASCAL's Design and PATH PASCAL User Manual," *SIGPLAN Notices*, **15**(9), pp. 13-24 (September 1980).

Chen78.

L. Chen and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation,'' *Digest of Papers FTCS-8: Eighth Annual Symposium on Fault-Tolerant Computing*, pp. 3-9 (1978).

Cohe85.

N. H. Cohen, ''Tasks as Abstraction Mechanisms,'' *ACM Ada Letters*, **V**(3-6), pp. 30-44 (November-December 1985).

Corn82.

D. Cornhill, J. Beane, and J. Silverman, *Distributed Ada Project*, Honeywell Systems & Research Center, Minneapolis, MN (1982).

Corn83.

D. Cornhill, ''A Survivable Distributed Computing System for Embedded Application Programs Written in Ada,'' *ACM AdaTEC Ada Letters*, **III**(3), pp. 79-87 (December 1983).

Cris82.

F. Cristian, ''Exception Handling and Software Fault Tolerance,'' *IEEE Transactions on Computers*, **C-31**(6), pp. 531-540 (June 1982).

Cris84.

F. Cristian, ''Correct and Robust Programs,'' *IEEE Transactions on Software Engineering*, **SE-10**(2), pp. 163-174 (March 1984).

Dijk68.

E. W. Dijkstra, ''Cooperating Sequential Processes,'' pp. 43-112, in *Programming Languages*, ed. I. Genuys, Academic Press, New York (1968).

Dijk76.

E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ (1976).

Eber83.

J. Eberhart, ''Message from Earth: Viking Phone Home,'' *Science News*, **123**(2), p. 20 (January 8, 1983).

Eswa76.

K. P. Eswaren, J. N. Gray, R. A. Lorie, and I. L. Traiger, ''The Notion of Consistency and Predicate Locks in a Database System,'' *Communications of the ACM*, **19**(11), pp. 624-633 (November 1976).

Garm81.

J. R. Garman, ''The Bug Heard 'Round the World,'' *ACM Software Engineering Notes*, **6**(5), pp. 3-10 (October 1981).

Gell78.

M. Geller, ''Test Data as an Aid in Proving Program Correctness,'' *Communications of the ACM*, **21**(5), pp. 368-375 (May 1978).

Good75a.

J. B. Goodenough and S. L. Gerhart, ''Toward a Theory of Test Data Selection,'' *IEEE Transactions on Software Engineering*, **SE-1**(2), pp. 156-173 (June 1975).

Good75b.

J. B. Goodenough, ''Exception Handling: Issues and a Proposed Notation,'' *Communications of the ACM*, **18**(12), pp. 683-696 (December 1975).

Gray76.

J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, ''Granularity of Locks and Degrees of Consistency in a Shared Data Base,'' pp. 365-394, in *Modelling in Data Base Management Systems*, ed. G. M. Nijssen, North-Holland, Amsterdam (1976).

Gray81.

J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, ''The Recovery Manager of the System R Database Manager,'' *ACM Computing Surveys*, **13**(2), pp. 223-242 (June 1981).

Groo82.

N. V. Groom and R. V. Hood, et al., ''Electric Flight Systems,'' CP-2209, NASA Langley Research Center, Hampton, Va. (February 1982).

Haer83.

T. Haerder and A. Reuter, ''Principles of Transaction-Oriented Database Recovery,'' *ACM Computing Surveys*, **15**(4), pp. 287-318 (December 1983).

Hans83.

R. C. Hansen, R. W. Peterson, and N. O. Whittington, ''The 3B20D Processor and DMERT Operating Systems: Fault Detection and Recovery,'' *The Bell System Technical Journal*, **62**(1), pp. 349-365 (January 1983).

Hech76.

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

Horn74.

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

Inge61.

P. Ingerman, "Thunks," *Communications of the ACM*, **4**(1), pp. 55-58 (1961).

Jalo84.

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

Jalo86.

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).

Kim82.

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).

Knig83.

J. C. Knight and S. T. Gregory, "On The Engineering Of Crucial Software," Final Report for NASA Grant Number NAG-1-233, Department of Applied Mathematics and Computer Science, University of Virginia, Charlottesville, Virginia (1983).

Knig85.

J. C. Knight, N. G. Leveson, and L. D. StJean, "A Large Scale Experiment in N-Version Programming," *Digest of Papers 15th IEEE International Symposium on Fault-Tolerant Computing Systems*, pp. 135-139 (June 1985).

Kohl81.

W. H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *ACM Computing Surveys*, **13**(2), pp. 149-184 (June 1981).

Lee78.

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

Lee80.

P. A. Lee, N. Ghani, and K. Heron, "A Recovery Cache for the PDP-11," *IEEE Transactions on Computers*, **C-29**(6), pp. 546-549 (June 1980).

Leve82.

N. G. Leveson, "Software Safety," *ACM Software Engineering Notes*, **7**(2), pp. 21-24 (April 1982).

Leve83.

N. G. Leveson and P. R. Harvey, "Analyzing Software Safety," *IEEE Transactions on Software Engineering*, **SE-9**(5), pp. 569-579 (September 1983).

Lies83.

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).

Lisk77.

B. Liskov, ''Abstraction Mechanisms in CLU,'' *Communications of the ACM*, **20**(8), pp. 564-576 (1977).

Lisk83.

B. Liskov and R. Scheifler, ''Guardians and Actions: Linguistic Support for Robust, Distributed Programs,'' *ACM Transactions on Programming Languages and Systems*, **5**(3), pp. 381-404 (July 1983).

Lome77.

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

Lome85.

D. B. Lomet, ''Making Pointers Safe In System Programming Languages,'' *IEEE Transactions on Software Engineering*, **SE-11**(1), pp. 87-96 (January 1985).

Lond71.

R. L. London, ''Software Reliability Through Proving Programs Correct,'' *Proceedings IEEE International Symposium on Fault-Tolerant Computing* (March 1971).

Long77.

A. B. Long, et al., ''A Methodology for the Development and Validation of Critical Software for Nuclear Power Plants,'' *Proceedings COMPSAC77*, pp. 620-626 (November 1977).

Lync83.

N. A. Lynch, ''Multiversion Atomicity-A New Correctness Criterion for Database Concurrency Control,'' *ACM Transactions on Database Systems*, **8**(4), pp. 484-502 (December 1983).

MacL77.

M. D. MacLaren, ''Exception Handling in PL/I,'' *ACM Sigplan Notices*, **12**(3), pp. 101-104 (March 1977).

MacL80.

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

Mell77.

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).

Merl78.

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).

Meye81.

F. Meyer, ''Computers Transform New Airliners,'' *High Technology*, **1**(1), pp. 73-78 (September-October 1981).

Myer76.

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

Myer78.

G. J. Myers, "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspection," *Communications of the ACM*, **21**(9), pp. 760-768 (September 1978).

Naur69.

P. Naur, "Programming by Action Clusters," *BIT (Nordisk Tidskrift for Informationsbehandling)*, **9**(3), pp. 250-258 (1969).

Neum80.

P. G. Neumann, "Letter from the Editor: More on the Reliability of Mankind-Critical Systems," *ACM Software Engineering Notes*, **5**(3), p. 2 (July 1980).

Neum81.

P. G. Neumann, "Letter from the Editor," *ACM Software Engineering Notes*, **6**(1), p. 1 (January 1981).

Neum82.

P. G. Neumann, "Letter from the Editor," *ACM Software Engineering Notes*, **7**(1), p. 1 (January 1982).

Neum83.

P. G. Neumann, "Letter from the Editor," *ACM Software Engineering Notes*, **8**(5), pp. 1-9 (October 1983).

Neum84.

P. G. Neumann, "Letter from the Editor," *ACM Software Engineering Notes*, **9**(1), pp. 3-7 (January 1984).

Parn72.

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

Rand75.

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

Russ77.

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

Russ79.

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

Salz78.

E. J. Salzman, "An Experiment in Producing Highly Reliable Software," in *M.Sc. Dissertation*, Computing Laboratory, University of Newcastle upon Tyne (1978).

Scot83.

R. K. Scott, J. W. Gault, and D. F. McAllister, "The Consensus Recovery Block," *Proceedings of the Total Systems Reliability Symposium*, pp. 74-85 (1983).

Shri78a.

S. K. Shrivastava, "Sequential Pascal with Recovery Blocks," *Software Practice and Experience*, **8**(2), pp. 177-185 (March 1978).

Shri78b.

S. K. Shrivastava and A. A. Akinpelu, ''Fault-Tolerant Sequential Programming Using Recovery Blocks,'' *Digest of Papers FTCS-8: Eighth Annual Symposium on Fault-Tolerant Computing*, p. 209 (June 1978).

Shri79.

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

Tane76.

A. S. Tanenbaum, ''In Defense of Program Testing or Correctness Proofs Considered Harmful,'' *ACM SIGPLAN Notices*, **11**(5), pp. 64-68 (May 1976).

Tayl80a.

D. J. Taylor, D. E. Morgan, and J. P. Black, ''Redundancy in Data Structures: Improving Software Fault Tolerance,'' *IEEE Transactions on Software Engineering*, **SE-6**(6), pp. 585-594 (November 1980).

Tayl80b.

D. J. Taylor, D. E. Morgan, and J. P. Black, ''Redundancy in Data Structures: Some Theoretical Results,'' *IEEE Transactions on Software Engineering*, **SE-6**(6), pp. 595-602 (November 1980).

Ulsa73.

E. Ulsamer, ''Computers - Key to Tomorrow's Air Force,'' *Air Force Magazine*, **56**(7), pp. 46-52 (July 1973).

VERk80.

"Contributions from the VERkshop on Formal Verification," *ACM Software Engineering Notes*, **5**(3), pp. 4-47 (July 1980).

VERk81.

"VERkshop II: Verification Workshop," *ACM Software Engineering Notes*, **6**(3), pp. 1-63 (July 1981).

Wei80.

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).

Well84.

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