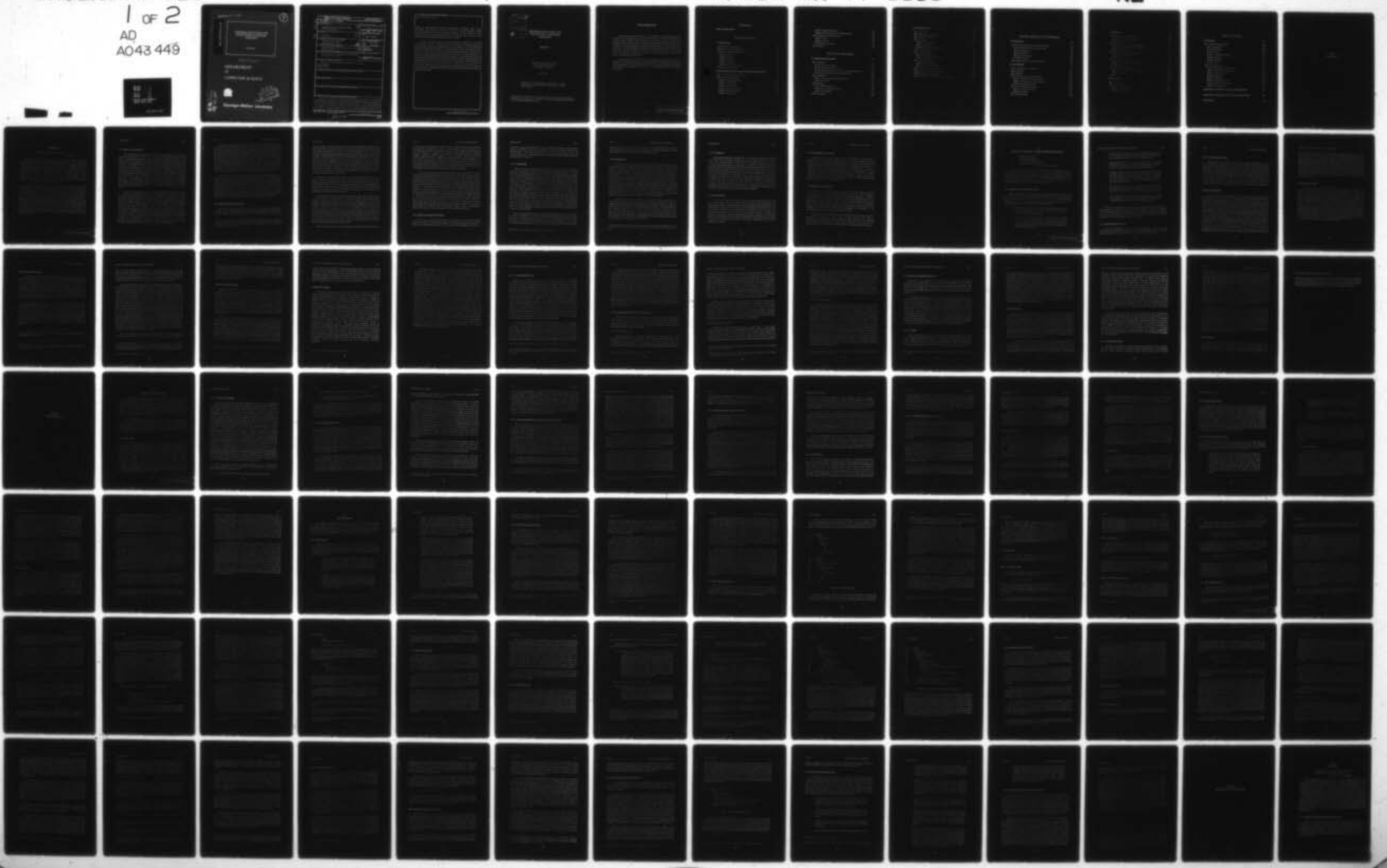
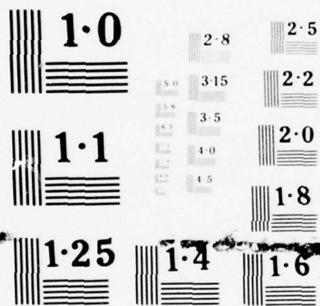


AD-A043 449
CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
PROGRAM STRUCTURES FOR EXCEPTIONAL CONDITION HANDLING.(U)
JUN 77 R LEVIN
F44620-73-C-0074
UNCLASSIFIED
AFOSR-TR-77-1136
NL
1 OF 2
AD
A043 449



NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

AFOSR-TR- 77-1136

7

ADA043449

PROGRAM STRUCTURES FOR EXCEPTIONAL CONDITION HANDLING

Roy Levin

Approved for public release;
distribution unlimited.

DEPARTMENT
of
COMPUTER SCIENCE

DDC
RECEIVED
AUG 29 1977
B.

AD No.
DDC FILE COPY

Carnegie-Mellon University

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 77-1136	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PROGRAM STRUCTURES FOR EXCEPTIONAL CONDITION HANDLING.		5. TYPE OF REPORT & PERIOD COVERED Interim rept.
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Roy Levin		8. CONTRACT OR GRANT NUMBER(s) F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept. Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, VA 22209		12. REPORT DATE June 1977
		13. NUMBER OF PAGES 76
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Air Force Office of Scientific Research (NM) Bolling AFB, DC 20332		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

158 p.

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

18. SUPPLEMENTARY NOTES

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

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

The goal of much of the current research in the field of "structured programming" is to develop a methodology that materially aids in the construction of clearly understandable yet rigorously verifiable programs. No single methodology has achieved pre-eminence, but some common threads run through all the serious contenders. To achieve clarity of expression, most methodologies postulate a small

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

403 081

20. ABSTRACT (Continued)

number of easily-described primitive programming constructs. (By "easily-described" we mean a construct has both a straight-forward intuitive characterization and a precise axiomatic definition of its semantics.) These constructs then become the building blocks for programs that exhibit an orderly arrangement of data and disciplined transfers of control.

To date, progress in this area has mostly been exhibited in successful "laboratory experiments". With a few notable exceptions (e.g. the New York Times system [Baker 72]), the demonstrations of the utility of such methodologies have been small-scale examples or toy systems. We contend that this is not an accident; we claim that, to date, programming methodologies have largely failed to address a crucial aspect of practical program construction: exception handling. The absence of methodologically sound language facilities for expressing behavior under exceptional circumstances has limited our ability to describe complex processing clearly and precisely. The goal of this thesis is to supply those missing facilities. In the words of the ancient cliche that introduces this chapter, we intend to test (prove) the "rule" of proposed programming methodologies to discover if they can be extended to handle both normal and exceptional cases with equal ease and clarity of expression.

ACCESSION for
NTIS White Section ☑
DDC Buff Section ☐
UNANNOUNCED ☐
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

PROGRAM STRUCTURES FOR EXCEPTIONAL CONDITION HANDLING

Roy Levin

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania

June, 1977

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was supported in part by the Defense Advanced Research Projects Agency under contract no. F44620-73-C-0074 and monitored by the Air Force Office of Scientific Research.

Acknowledgements

It is a pleasure and a privilege to thank Bill Wulf, my principal advisor and mentor, for his untiring interest and support. Without his insistence I might still be chasing bugs in Hydra. The collective criticisms and comments of the other members of my thesis committee - Peter Hibbard, Mary Shaw, and Alice Parker - and of Anita Jones have significantly enhanced my understanding of exception handling problems. I am also indebted to Paul Hilfinger, Dave Jefferson, Ralph London, and Larry Flon, on whose expertise in the field of program verification I have often relied. Several discussions with Fred Pollack and Dan Siewiorek have helped focus the specific proposals of this thesis.

I am grateful to my fellow Hydrants Hank Mashburn and Sam Harbison, whose skillful ministrations to the kernel enabled me to find the time to pursue this work. The appearance of this document is the result of a series of incantations performed by Brian Reid, a veritable woods wizard in the enchanted forest of document production.

PRECEDING PAGE BLANK-NOT FILMED

CONTENTS

Acknowledgements iii

Part I: Preliminaries

1 Introduction 3

- **1.1** What is an Exception? 4
- **1.2** Motivation and Background 5
- **1.3** Goals and Scope of the Thesis 7
  - **1.3.1** Verifiability 8
  - **1.3.2** Uniformity 9
  - **1.3.3** Adequacy 10
  - **1.3.4** Practicality 10
  - **1.3.5** Summary of Properties 11
- **1.4** Synopsis of the Thesis 11

2 Survey of Existing Exception Handling Mechanisms 13

- **2.1** Some Exception Handling Issues 13
- **2.2** Sequential Mechanisms 14
  - **2.2.1** Unusual Return Value 15
  - **2.2.2** Forced Branch 15
  - **2.2.3** Non-local GoTo 16
  - **2.2.4** Procedure Variables 17
  - **2.2.5** PL/I ON Conditions 19
  - **2.2.6** BLISS Signal 20

2.2.7 The MPS Mechanism 22
2.2.8 Some Proposed Sequential Mechanisms 23
2.3 Parallel Program Mechanisms 26
2.3.1 Polling 26
2.3.2 Interrupts 27
2.3.3 Message Systems 28
2.4 Summary 29

Part II: A New Mechanism

3 *Elaboration of the Goals* 33
3.1 Uniformity 33
3.1.1 Choice of Language 34
3.1.2 Propagating Exceptions 35
3.1.3 Associating Detection and Processing of Exceptions 37
3.1.4 The Role of the Exception Mechanism 39
3.2 Verifiability 40
3.2.1 Using Existing Language Properties 41
3.2.2 Predicates 42
3.3 Adequacy 43
3.3.1 Memory Data Error 44
3.3.2 Resource Allocation Failure 44
3.3.3 I/O Completion 45
3.4 Practicality 46

4 The Mechanism

4 The Mechanism 49

4.1 Terminology 49

4.2 Relevant Language Notions 51

4.3 A Gentle Introduction 53

4.4 Conditions 56

4.4.1 Condition Names 56

4.4.2 Parameters 57

4.4.3 Conditions and Instances 57

4.5 Handler Definition 59

4.6 Handler Eligibility 65

4.6.1 The Eligibility Rule 66

4.6.2 Another View of 'Contexts' 71

4.7 Raising Conditions 72

4.7.1 Selection Policies 72

4.7.2 The Raise Statement 78

4.8 Handler Invocation Semantics 79

4.9 Handler Termination Semantics 81

4.10 An Informal Recapitulation 83

4.11 A Postscript: Synchronization Esoterica 85

Part III: Justification of the Mechanism

5 Uniformity 89

5.1 Interactions with the Embedding Language 89

5.1.1 Variable Access and Scope Rules 90

5.1.2 Shared Data 91

5.1.3 Parallelism 92

5.1.4 Synchronization 93

5.1.5 Protection 96

5.2 Simplifying the Use of the Mechanism 96

6 Verifiability 99

6.1 Assumptions and Conventions 99

6.2 Predicates 101

6.3 Notation 103

6.4 Proof Rules for the Signalling Site 104

6.4.1 The Broadcast-and-Wait Policy 104

6.4.2 The Sequential-Conditional Policy 106

6.4.3 The Broadcast Policy 107

6.5 Proof Rule for Handler Sites 108

6.5.1 Transformation Rules 108

6.5.2 Proof Rule 110

6.6 An Assessment 112

7 Adequacy 115

7.1 Example 1: Symbol Table 116

7.1.1 The Symbol Table Problem 116

7.1.2 Assessment 120

7.2 Example 2: Inconsistent Data Structures 121

7.2.1 The Inconsistent String Problem 121

7.2.2 Assessment 124

7.3 Example 3: Arithmetic Exceptions 124

7.3.1 Floating-Point Underflow 125

7.3.2 Assessment 127

7.4 Example 4: Resource Allocation 128

7.4.1 The Storage Allocation Problem 128

7.4.2 Assessment 132

7.5 Example 5: I/O Completion 133

7.5.1 The Real-Time Update Problem 133

7.5.2 Assessment 135

7.6 Summary 136

8 Practicality 137

8.1 Handler Bodies 137

8.2 Eligible Handlers Set 138

8.2.1 Enabled Handlers 139

8.2.2 Eligible Handlers 141

Part IV: Conclusion

9 *Summary* 149

9.1 Contribution of this Work 149

9.1.1 Specification 149

9.1.2 Abstraction 150

9.1.3 Sharing 150

9.1.4 Programming Flexibility 151

9.1.5 Language 151

9.1.6 Verification 152

9.1.7 Cost 152

9.2 Remaining Issues 153

9.2.1 Selection Policy Primitives 153

9.2.2 Verification of Synchronization 153

9.2.3 Usage Paradigms 154

9.2.4 Protection 154

9.2.5 Enforcement 154

9.2.6 Hardware Applicability 155

9.2.7 Uniform Control Structure 155

9.2.8 Self Applicability 156

Appendix A: A More Flexible Handler Definition 157

Appendix B: The Alphard Verification Methodology 159

References 163

Part I
Preliminaries

1
Introduction

"It's the exception that proves the rule."

- Old chestnut

The goal of much of the current research in the field of "structured programming" is to develop a methodology that materially aids in the construction of clearly understandable yet rigorously verifiable programs. No single methodology has achieved pre-eminence, but some common threads run through all the serious contenders. To achieve clarity of expression, most methodologies postulate a small number of easily-described primitive programming constructs. (By "easily-described" we mean a construct has both a straight-forward intuitive characterization and a precise axiomatic definition of its semantics.) These constructs then become the building blocks for programs that exhibit an orderly arrangement of data and disciplined transfers of control.

To date, progress in this area has mostly been exhibited in successful "laboratory experiments". With a few notable exceptions (e.g. the New York Times system [Baker 72]), the demonstrations of the utility of such methodologies have been small-scale examples or toy systems. We contend that this is not an accident; we claim that, to date, programming methodologies have largely failed to address a crucial aspect of practical program construction: exception handling. The absence of methodologically sound language facilities for expressing behavior under exceptional circumstances has limited our ability to describe complex processing clearly and precisely. The goal of this thesis is to supply those missing facilities. In the words of the ancient cliche that introduces this chapter, we intend to test (prove) the "rule" of proposed programming methodologies to discover if they can be extended to handle both normal and exceptional cases with equal ease and clarity of expression.

PRECEDING PAGE BLANK-NOT FILMED

1.1 *What is an Exception?*

We cannot give a terse definition of 'exception' that would adequately serve the intent of this thesis. To claim an exception is a rarely occurring event merely begs and distorts the question. Frequency only has meaning in a relative sense, and the same event may occur relatively often in one context and relatively rarely in another. As a simple example, consider the operation that looks up a symbol in a compiler's symbol table. It has two obvious possible outcomes: 'symbol present' and 'symbol absent'. Although we tend to view the former as 'normal' and the latter as 'exceptional', this is purely a psychological prejudice. If the 'lookup' operation is used to determine the interpretation of a symbol in a program body (as opposed to the declarations), then 'symbol absent' probably corresponds to 'undeclared identifier' - an exceptional condition in most cases. If, however, the 'lookup' operation is used during declaration processing, then 'symbol absent' is probably the 'normal' case and 'symbol present' (i.e. 'duplicate definition') the exception. What constitutes an exception evidently depends on the context in which an event occurs.

Context is not the only factor - the language/system in which computations are expressed markedly influences the designation of exceptions. Some structures, notably production systems [Rychener 76] and Dijkstra's guarded commands [Dijkstra 76], encourage the programmer to define the normal case and the exceptions with the same syntax. Such 'event-driven' systems obviously have no need for special language constructs to express exception handling behavior. Yet they are not ideal in some respects. Because all events are represented similarly, a reader of a program may experience difficulty in separating the primary computation from the special cases. Clarity therefore may be sacrificed. Even if it is not, production systems may not provide the appropriate structure for many practical tasks, which often require the more conventional facilities of a procedural language. Procedural languages, historically, have provided few constructs for exception handling (see chapter 2) and will therefore be the primary vehicle for the mechanism proposed in this thesis.

Exceptions are sometimes called 'errors', but we reject the connotations of that term and strenuously avoid its use. An 'error' is usually an event whose occurrence, though not unexpected, is not essential to the correct completion of the primary computation, and may be detrimental. We prefer to take a broader view that includes such 'errors' as a proper subset of exceptions, allowing (relatively) infrequent events to qualify as exceptions as well.[1] In doing so we deliberately allow exceptions to spill over into the area of communication facilities. For example, the indication that an I/O operation has completed may be viewed as either an exception (relatively infrequent event) or a communication signal, depending upon the usage context. We consider the existence of this fuzzy boundary to suggest robustness, not sloppiness, in an exception handling mechanism. A sharp line of demarcation would probably limit the flexibility of programs operating near it.

We are led, therefore, to ask what distinguishes exception handling from more general communication between program units. Again, because of the overlap, we cannot make a sharp separation. Generally, the use of an exception handling mechanism is preferable when a programmer wishes to 'play down' the special-case processing in a particular context. An exception mechanism should be able to limit the visibility of such processing, suppressing detail where desirable, so that the primary computation can be stressed. It may also distribute the costs rather differently than a general communication facility does. Thus the choice of an exception or communication mechanism may be a matter of the programmer's taste, assuming, of course, that both provide adequate function at acceptable cost.

1.2 Motivation and Background

Why worry about exception processing? Anyone who has ever built a large software system or tried to write a 'robust' program can appreciate the problem. As programs grow in size, special cases and unusual circumstances crop up with

[1] If we look at the evolution of programming, we see that the original source of exceptions that programs sought to handle was hardware errors. The techniques employed were gradually codified in higher-level languages, but though they have been extended in some ways, the mechanisms most languages provide are just the "fall-out" from those earliest error-handling attempts.

startling rapidity. Even in moderate-sized programs that perform seemingly simple tasks, exceptional conditions abound. Consider a tape-to-tape copy program. Any reasonable programmer will handle an end-of-file condition, since it probably indicates completion of the copying operation. But what about tape errors? End-of-tape? Hung device? Record larger than expected? We could enumerate other possible exceptions, but the point is clear. Exceptions exist even in the simplest problem, and the complexity they induce in large programs can be mind-boggling. We contend that no mechanism has yet been implemented or proposed that can control this complexity. A look at the (dis)organization of existing large systems should easily convince us that such control is essential if we ever hope to make these systems robust, reliable, and understandable. This thesis provides an exception mechanism equal to that task.

Although it is obvious that any exceptional condition that arises must be handled if our programs are to be robust, we might wonder whether we need a single, general mechanism to do so. Why not simply test explicitly for an exception at all possible points in the program where it can occur? If this is prohibitively expensive or inconvenient, why not test only at a selected subset of these points? No special mechanism is required here, and the code to detect these exceptions is explicit and under the programmer's control.

The objections to this ad hoc approach should be clear. For some classes of exceptions (e.g. I/O completions), the condition may occur virtually anywhere in the program. Obviously, it is impractical to include an explicit test "at all possible points" where such exceptions can arise. Polling at "selected" points may be feasible in principle, but in practice destroys the structural coherence of the source program. Because of timing considerations, it often becomes necessary (with a polling scheme) to introduce tests for exceptions into pieces of the program that have nothing to do with the condition being tested. It is then impossible to read and understand such a program segment without understanding the entire structure of which it is a (perhaps very small and localized) part. Explicit polling may suffice in very limited applications but is clearly inadequate for general use. A technique must be found that preserves structural clarity.

Once we begin to search for a more suitable exception handling technique, we

find it useful to subdivide the problem. We may view the process of handling an exception as consisting of three phases: *detection* of the condition, *transmission* of the notification of its existence, and *diagnosis* and *recovery* from the (implied) problem. Detection, diagnosis, and recovery are exception-specific, although common techniques may be applied for many exceptions. Indeed, there exist methodologies that seek to address and codify these techniques [Pollack 77]. This topic is beyond the scope of this thesis; we will concentrate our attention on the transmission of exception notifications. It will be our goal to provide a general mechanism to pass information about an exception between the point of its detection and the points at which it may be processed intelligently.

The historical motivation for this work should be briefly mentioned. For many years I have been annoyed with the incompleteness of the specifications of various (software and hardware) facilities I have attempted to use. The problems arise not in understanding the 'normal case' behavior of a facility - that is (nearly) always defined clearly. Discovering the behavior under unusual circumstances is another matter, since the author of the specifications has rarely bothered to ask himself: "What if ...". Consequently, the specifications, and often the facility as well, do not address exceptional conditions. I became painfully aware of the difficulty of defining exceptional behavior in a large system during the design and construction of the Hydra kernel [Wulf 74]. The implementation language available (BLISS-11 [DEC 74]) did not provide sufficiently powerful tools for exception handling, yet a significant fraction of the operating system code was devoted to detecting, transmitting, and processing exceptions. This experience convinced me that a general mechanism was needed that could serve as the basis for the precise specification of exceptions. But what properties should it have? In seeking to answer that question, I embarked upon the work reported in this thesis.

1.3 Goals and Scope of the Thesis

For reasons suggested above, we will concentrate on exception handling facilities in a generally procedural language. The precise syntax and semantics of such a mechanism will necessarily be influenced by the language and system in

which it is embedded. Nevertheless, we can identify desirable properties that a general mechanism should exhibit before it can be considered 'eligible' for implementation. These properties will serve as goals to be attained by our proposed mechanism. We briefly present these goals here; they will be explored in greater depth in chapter 3.

1.3.1 Verifiability

Any proposal for a new language mechanism should be closely scrutinized to determine the effect of the proposed facility on program structure. Existing mechanisms frequently exhibit glaring inadequacies in this respect, and, as a result, induce complex and confusing program structures. It is difficult to require clarity and measure it quantitatively, which perhaps accounts for much of the current debate on language constructs to support structured programming. We maintain that clarity in an exceptional condition mechanism is of the utmost importance, for the following reason. If we consider 'robust' programs (i.e. those that handle most of the exceptional conditions that can occur in their execution environment), we find that a large fraction, if not an actual majority, of their code is devoted to exceptional condition handling.[2] Recalling the oft-quoted observation that a program spends 90% of its execution time in 10% of its code, we would expect to find exceptional condition handling code concentrated in the infrequently executed 90% of the program text. Both the high proportion of exceptional condition handling code and the infrequency of execution argue strongly for language mechanisms that strive for clarity. But since clarity is impossible to guarantee in practice, we demand instead verifiability, believing that constructs that are easily verified are generally easily understood.

We will require that a general exceptional condition mechanism be amenable to formal verification at a level consistent with the other constructs in the language. This implies that the essential (i.e. language independent) semantics of the mechanism must observe program structuring principles that have been shown to facilitate verification, e.g. locality of reference to data structures and disciplined

[2] Operating systems are perhaps the most common example of such programs.

transfer of control. Of course, the adaptation of the mechanism for a specific implementation must also respect these principles. Verifiability will be an essential property of a well-structured exception handling mechanism.

1.3.2 Uniformity

A fundamental problem with existing mechanisms is that they provide an acceptable means of handling only limited classes of exceptions. When a large software system is constructed using several such mechanisms, the interfaces between different parts become awkward, inconsistent, difficult to understand, and consequently error-prone. A general mechanism permits a pleasing uniformity of expression at all levels of the system and for all exceptions, facilitating understanding and, incidentally, simplifying verification conditions.[3] We want our exceptional condition handling mechanism to be applicable at all levels in a software system, from user program to operating system/hardware interface. (This may force a substantial change in the way we view peripheral devices, but if it permits a uniform approach, it is worthwhile.) We may be compelled, by the limitations of physical implementation, to require only that hardware provide a subset of the general mechanism's facilities; however, that subset in itself will be consistent with the other goals of the mechanism, namely verifiability and adequacy.

It is important to note that uniformity in application of the mechanism does not constrain us to live with a single implementation at all levels and for all conditions. Indeed, the implementation of (the subset of) the general mechanism in hardware will obviously be different from the implementation in software. The same holds true for operating system and user; the implementations of a single conceptual facility may, and probably will, differ. It may even be possible that several distinct implementations will exist within a single (large) piece of the system (this is true, for example, in the kernel of Hydra [Wulf 74]). We will require uniformity only in the functional specification of the mechanism.

[3] We might extend this argument and propose a single, unified syntax and semantics for general control flow. Indeed, this is done in production systems [Rychener 76]. Our reasons for not doing so are discussed in section 9.2.7.

1.3.3 Adequacy

While we may define a mechanism that is uniform at all system levels and that lends itself easily to formal verification, we do not thereby insure that it is useful. To be worth implementing, the mechanism must be capable of solving a variety of 'real world' problems naturally. This last word, "naturally", is the fly in the ointment, since it implies an aesthetic judgment of a particular problem formulation. We do not hope to prove that the proposed mechanism is 'complete', (that is, that it will solve all exceptional condition handling problems.) Even if we could do so, we could not prove that it would do so "naturally". Accordingly, we settle for a notion of 'adequacy', far weaker than completeness, which merely shows that the mechanism is indeed good for something. We will demonstrate 'adequacy' inductively by example rather than deductively by proof. However, as a specific condition on adequacy, we will demand that the mechanism accommodate parallel as well as sequential programming, for to restrict it to one or the other is to exclude a major source of 'real world' exceptional condition handling problems.

1.3.4 Practicality

It is not sufficient merely to define a mechanism whose functionality makes it worth implementing. We must also be able to deliver that functionality with reasonable efficiency. No programmer will take advantage of the mechanism if the implementation makes it too costly to use. We must supply the intended function at a cost that is attractive to the programmer, for otherwise he may employ other language constructs to accomplish the exception handling function ... and sacrifice clarity in the process. While the programmer need not, in general, be acquainted with the implementation of the exception mechanism, he is entitled to know that it supplies the advertised function at a cost that is competitive with alternate language constructs.

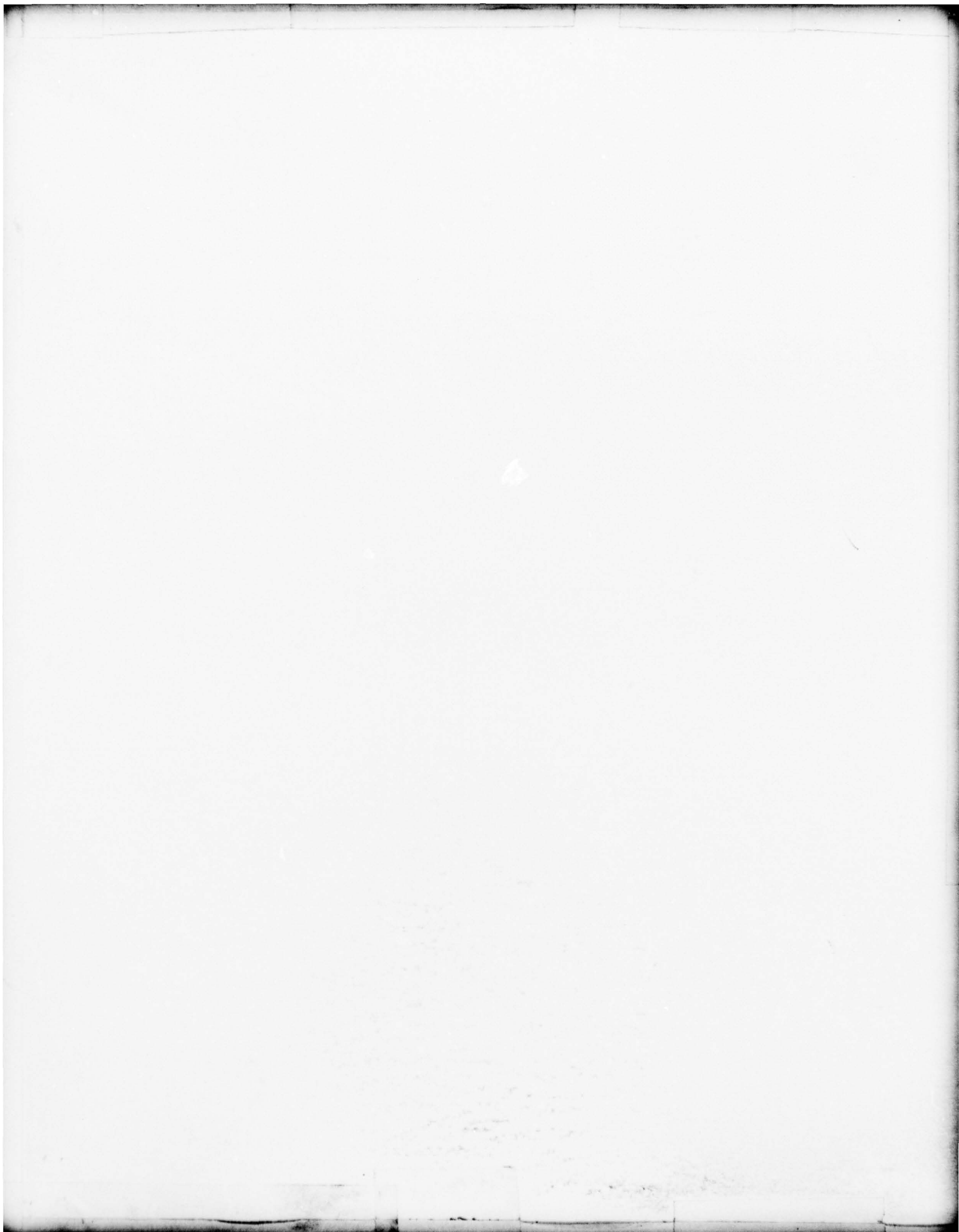
1.3.5 Summary of Properties

These properties, then, characterize the fundamental requirements for a general exceptional condition handling mechanism. We assert that a mechanism exhibiting these properties represents an advance in the state of the art, since no existing mechanism meets these requirements. (The supporting evidence is in chapter 2). We are requiring that our new mechanism be able to solve a collection of practical problems that heretofore have been handled only by a variety of *ad hoc* techniques. Furthermore, we require that it do so not by a jumble of loosely-integrated facilities, but by a single, uniform mechanism that can be formally proved to behave in precisely specified ways. Finally, it must be implementable with low enough cost to be of practical utility.

1.4 Synopsis of the Thesis

We cover the topic of exception processing in four stages, which correspond to the four major parts of the thesis. Part I introduces the topic in two chapters. This chapter has established goals for the remainder of the work. Chapter 2 surveys previous approaches to exception processing and evaluates them in terms of the goals of the preceding section. With this background established, part II proceeds to a presentation of a new mechanism. First, chapter 3 elaborates the goals and explores their implications on the structure of the mechanism, then chapter 4 presents the details of that mechanism, concentrating on an operational specification in language-independent terms.

Part III demonstrates that the mechanism of part II meets the goals laid down in part I. Chapters 5, 6, 7, and 8 each address one goal in detail, relating the specifics of the proposed mechanism to the concerns embodied in the general statements above. Finally, part IV summarizes the content of this work and considers the remaining unsolved problems and possible future solutions.



2
Survey of Existing Exception Handling Mechanisms

My object all sublime
I shall achieve in time -
To let the punishment fit the crime -
The punishment fit the crime.

- W. S. Gilbert, The Mikado

To acquire some perspective on the state-of-the-art in exceptional condition handling, we will examine programming language and system mechanisms that have been used or proposed for this purpose. Naturally, our survey does not include every language or construct that has attempted to address the problem, but it does cover the *techniques* that have been applied. We include in our list a few mechanisms that have been proposed but not implemented.

2.1 *Some Exception Handling Issues*

Before embarking on a survey of existing exception handling techniques, we briefly enumerate some issues that will arise frequently in subsequent discussions. We do not attempt here to explore these issues in depth; we merely list them as questions to be addressed both by the existing mechanisms described in this chapter and the proposed mechanism presented in chapter 4.

* *Specification* - How are exceptional conditions defined? Are their semantics formally specified? What is the space of situations to which they may apply?

* *Abstraction and Program Structure* - How does the exception mechanism aid in abstraction? Does it aid in the decomposition of programs along abstraction boundaries? Is it possible to process an exception without knowing the representations employed by the program that detects it?

PRECEDING PAGE BLANK-NOT FILMED

* *Sharing* - Does the mechanism recognize shared abstractions? If an abstraction is shared among several contexts, can exception processing related to that abstraction be shared as well?

* *Programming Flexibility* - To what extent does the exception mechanism influence non-exceptional processing? Does it restrict the set of permissible control and data structures?

* *Language* - What restrictions and assumptions does the exception mechanism impose upon the language in which it is embedded? Are its requirements relatively general or highly language-specific? How closely is it bound to global language properties, e.g. sequential processing?

* *Verification* - How does the mechanism affect the verifiability of its embedding language? In what ways does the desire for verification constrain the semantics of the exception mechanism?

* *Cost* - What is the cost of using the mechanism, and how is that cost distributed? Is the mechanism competitive with potentially attractive alternate constructs in the embedding language?

This list is by no means exhaustive, but it samples the most important issues. The questions are substantial ones, and are often directly or subtly inter-related. In the survey that follows, we will not attempt to examine each issue in the light of each mechanism; rather, we will try only to assess the prominent strengths and weaknesses of each mechanism in terms of this list.

2.2 Sequential Mechanisms

By far the majority of exception handling mechanisms address only the problem of sequential programming, for obvious historical reasons. We will consider the development of sequential mechanisms in roughly chronological order.

2.2.1 Unusual Return Value

Perhaps the most primitive form of exceptional condition handling mechanism is the 'unusual return value' from a subroutine invocation, indicating that some abnormal event has occurred. This technique is as old as the subroutine concept itself, and is probably best viewed as a degenerate form of exceptional condition mechanism. Its obvious deficiencies include cost (the condition must be explicitly checked and 'passed up' at all program levels), poor abstraction characteristics (it tends to lead to programming puns that are based on internal data structure representations within the computer), and imprecise specification. While the unusual return value has some utility in very localized contexts, in general it is heavily over-used and is practically a cliche among exception handling constructs.

2.2.2 Forced Branch

The simplest form of exception handling mechanism that actually involves a non-standard control structure is the forced branch. In assembly language this mechanism frequently appears as a 'skip return' from a subroutine, i.e. the instruction immediately following the subroutine call is or is not skipped to indicate the presence or absence of an exceptional condition. In FORTRAN IV and ALGOL 60 this mechanism is generalized to permit several labels to be passed as parameters to a subroutine, such labels being treated as possible alternate return points. The central notion is that the callee detects an exceptional condition and performs some *fixed* action (e.g. add one to the return address and exit, or branch to the location specified by the third parameter), which is used by the caller to initiate a condition handler. Such mechanisms generally impose little or no overhead in the 'normal return' case, and thus eliminate the cost of explicit checking inherent in the unusual return value. However, they raise serious program structuring issues. Failure by a caller to provide a legitimate handler for every condition the callee can signal may lead to unpredictable or catastrophic results at run-time. (In some FORTRAN IV implementations, for example, omission of a second label parameter to a

subroutine that expects three does not result in a compile-time error, and, at run-time, may lead to the handler for the *third* alternate return being invoked when the second was intended ... and without any error message.) Also, the use of simple labels to specify exceptional condition handlers (instead, for example, of subroutine names) encourages programmers to merge where possible the code for such handlers with the main-line program and other handlers, in a mistaken attempt at 'efficiency'. It then becomes extremely difficult to determine how many call sites reference a particular piece of handler code and what the possible program states can be on entry to that handler. These undisciplined transfers of control complicate the task of verifying the correctness of a subroutine in much the same way as the general goto does.

2.2.3 Non-local GoTo

A similar (though less desirable) construct, which appears in ALGOL 60 and some other block-structured languages is the non-local goto (i.e. a goto that specifies a destination outside the procedure in which it appears). This device has little to recommend it on program structuring grounds, for it gives the caller very little control over the choice of handler and further blurs the distinction (in the calling program) between main-line and exceptional condition handling code. In addition, although the non-local goto permits the exception notification to pass outside the detecting procedure (a desirable property), it forces the handler (i.e. the target label) to appear in a lexically enclosing block. Such a structure is rarely appropriate; the calling block (which may not be lexically enclosing) can normally supply a more meaningful handler. Furthermore, the non-local goto may abort the calling procedure without giving it an opportunity to "clean-up" its data structures. This undisciplined transfer of control exhibited by the general goto is a source of many verification difficulties in languages that possess it.

2.2.4 Procedure Variables

Languages that permit variables to refer to procedures contain the basis of a primitive exception mechanism. Algol-68 [van Wijngaarden 75] provides the most completely developed example of this approach. In Algol-68, input/output occurs on 'files', which are represented by a data structure that, among other quantities, contains a number of procedure variables.[1] Each such variable is associated with a particular exceptional condition, e.g. logical end-of-file. When an exception arises, the appropriate procedure variable is used to locate a procedure that is to be invoked in order to process the condition. Depending on the (boolean) value returned by the procedure, the file manipulation procedure that detected the exception either completes or "takes some sensible action".

This association of procedures with objects has been called "object-oriented exception handling" [Goodenough 75] and is not unique to Algol-68 (see, e.g. [Ross 67]). There are a few twists, however, to the Algol-68 approach that are worth noting. Since a file in Algol-68 is a 'structured value' (a 'record' in the sense of Algol-W [Wirth 66] and a number of other languages), one might expect that a user of a file could alter the "exception handling procedures" by storing into the appropriate fields of the 'file' data structure. However, Algol-68 does not permit direct user access to a 'file', and provides instead 'event routines' that perform store operations into the file's procedure variables. Thus instead of writing

```
logical file end of my data file := eof handler
```

where 'eof handler' is a procedure name and 'logical file end' is the relevant field of 'my data file', one must write

```
on logical file end(my data file, eof handler)
```

[1] Algol-68 does not use the term 'procedure variable'; rather, the notion is a special case of the more general concept of 'mode'. In this discussion we will substitute more conventional, though less precise, names for the possibily unfamiliar terminology of Algol-68.

Here, the event routine 'on logical file end' performs the desired association. We prefer the latter form as well, but for structuring rather than efficiency reasons. By forcing the use of event routines, Algol-68 is, in effect, preventing direct external manipulation of the 'file' data structure. The representation of a 'file' can then be freely changed without necessitating changes in the programs that use it.[2]

Evidently, procedure variables are a powerful language facility that can be pressed into service as an exception handling mechanism. However, as formulated in Algol-68, they are not ideal for this purpose. First, the Algol-68 scope rules require that a procedure's scope be no smaller than the scope of the procedure variable to which it is assigned. In terms of our example above, the scope of 'eof handler' must encompass the scope of 'my data file'. While this stringent requirement helps prevent dangling pointers, it makes practical exception processing rather inconvenient, since it limits our ability to construct procedures that will handle the exception only in a very localized context. Second, only a single procedure is ever invoked in response to an exception, though there may be many contexts in which recovery actions may (should) be attempted.[3] Although we can build composite structures that hold more than one procedure variable, we do not have, in Algol-68, adequate information to maintain this structure without an undue amount of user assistance. In seeking to acquire that information (e.g. the contexts in which the user is executing when the exception is detected), we are again stymied by the rigid scope rules. Third, procedure variables make it difficult to discover exactly what action will be taken when a particular exception occurs. The body of exception processing code may be (lexically) far removed from the invocation of a function that triggers it. Clarity is essential (see section 1.3) and should be encouraged in an exception mechanism.

We should not overlook the desirable properties of the Algol-68 approach. Exception handlers are procedures and thus return to their invokers, permitting the

[2] It is noteworthy that the final report on Algol-68 [van Wijngaarden 75] introduces event routines purely to permit efficient implementation of transput. (The original report [van Wijngaarden 69] had no such notion; exception handling procedures were altered by direct assignment.) For structures defined by an Algol-68 user, the component fields _cannot_ be made inaccessible and thus the use of routines to alter the fields cannot be enforced.

[3] We will discuss this issue in detail in chapter 4.

detector of the condition to respond to the recovery action taken.[4] Handlers, being general procedures, are parameterized, and communication through global variables is therefore unnecessary (in contrast with PL/I ON conditions, below). Finally, handlers are associated with the object rather than the operation - an important step in the direction of proper abstraction. The mechanism of chapter 4 will build on all of these properties.

2.2.5 PL/I ON Conditions

We now come to mechanisms that provide a means of passing over dynamic program levels. The PL/I ON condition mechanism [IBM 70] was perhaps the first attempt to provide a true exceptional condition mechanism in a general purpose high-level language. 'Conditions' are explicitly declared (actually, they are called 'events'), and triggered by a special statement. Handlers are also explicitly specified and associated with particular conditions. A number of built-in conditions (e.g. arithmetic overflow, end-of-file, conversion error) are signalled by the implementation, but may be handled by user-supplied handlers in much the same way as programmed conditions. When a condition is raised, the most recently encountered handler in the dynamic nesting of program blocks (usually) receives the notification.

Although the basic notion behind ON conditions is sound, it does not mesh well with other language facilities and seems to have been conceived largely as a means to allow programs to trap a collection of system-defined run-time errors. While it attempts to accommodate programmer-defined conditions and even parallel processing (within an inherently hierarchical process structure), the foundations are weak and the mechanism remains awkward for all but the original specialized conditions. Specifically, ON conditions suffer from reliance upon the non-local goto, absence of parameters to handlers, and a glaring lack of uniformity in the actions handlers are permitted to take. We have already considered the shortcomings of the non-local goto. The absence of parameters to handlers forces them to

[4] Assuming, of course, that the exception handling procedure does not perform a non-local goto. The evils of this construct have already been examined.

communicate with the signaller through shared global variables. This undisciplined access to data engenders many of the same verification problems that the general goto does. In short, the weaknesses of the ON condition mechanism overshadow its desirable properties. The primary contribution of this first attempt at high-level exception handling has been its influence on subsequent systems (e.g. BLISS [DEC 74], MPS [Lampson 74]) that built on the ON condition notion.

2.2.6 BLISS Signal

The BLISS 'signal-enable' construct [DEC 74] is perhaps the simplest means of passing over dynamic program levels.[5] Though the actual mechanism is slightly more general than described here, signal-enable provides, in essence, an explicit means of defining an exceptional condition and supplying a handler to respond to its detection. The handler is defined by an enable declaration, which names the condition and specifies a piece of program text to be executed when the condition is raised via a signal statement. Handlers are associated with a lexical block (in the ALGOL sense), but are maintained at run-time in a LIFO stack, so that the most recently encountered handler (in the *dynamic* block nest) for a signalled condition will be invoked. A handler itself is allowed to signal, which permits propagation of the signal to a higher-level block. Normal (i.e. non-signalling) termination of a handler terminates the block in which the handler is defined. There is an important additional form of enable that defines a handler capable of processing any signalled condition. The signal-enable mechanism provides no facilities for parameter passing (from signaller to handler) and does not permit resumption of the signaller upon completion of the handler's actions. Furthermore, no 'passed over' intermediate program levels are informed of the signalled condition's occurrence; these blocks must include appropriate enable declarations for exceptional conditions which may affect their execution. In spite of these weaknesses, the explicit definition of handlers and special syntax for signalling conditions represent important advances over the mechanisms previously mentioned.

[5] A functionally similar mechanism is described in [Bron 76].

The BLISS mechanism, while adequate for many applications, has some shortcomings. The inability to pass parameters from signaller to handler leads to awkward global variable usage, which in turn destroys the potential for well-modularized programs. Also, because it does not permit a handler to return to the program context in which the signal occurred, this mechanism is only useful for 'fatal' errors; that is, conditions that cannot be corrected by the handler without invalidating the interrupted computation. The BLISS mechanism, in keeping with the philosophy of the language, is designed to provide substantial function at minimal cost, but as these two limitations suggest, it is not suitable for all exceptional condition handling situations. As an example, consider the position of a general storage allocator in a complex system. If it is unable to satisfy an allocation request using its own resources, it will raise the 'out-of-storage' condition, requesting other system modules to release some storage. It cannot make this request via a BLISS signal since the current invocation of the 'get-storage' routine will be terminated, and the allocator will then have no opportunity to satisfy the request even if some 'out-of-storage' handler releases adequate space. The allocator is forced to use an explicit subroutine call, implying that it knows the names of the routines (in effect, handlers) in other system modules that can potentially release storage. But this suggests that every time a new module that uses the allocator is introduced into the system, an additional routine call should be included in the allocator to invoke this new module's 'try-to-free-some-storage' routine. We see what violence this does to the specifications of the allocator by introducing an awkward circularity. By allowing resumption of the signaller upon handler termination, we can eliminate many such structuring problems. This leads to the MPS mechanism, which we examine next.

2.2.7 The MPS Mechanism

MPS [Lampson 74][6] solves the problems inherent in the BLISS mechanism[7] by treating handler invocations as subroutine calls made from the signaller's context. The important distinction from normal subroutine calls is that the signaller does not know what 'subroutine' he is invoking. In fact, MPS specifies that an ordered list of handlers be consulted, with each one being passed the condition name and an argument record (in the ALGOL-W [Wirth 66] sense) composed of parameters supplied by the signaller. If a handler 'rejects' the signal, the next handler in the list is consulted, but if a handler corrects the condition, it may resume execution of the signaller by a simple subroutine return (including a return record, if desired). Finally, the handler may raise the system signal unwind, which implies that the (original) signaller and all 'passed over' program levels are about to be terminated. Each such level has an opportunity to process the unwind signal (by providing an unwind handler), after which the level is deleted. This mechanism permits a more natural structure for the storage allocator example, one that eliminates several of the difficulties mentioned above. It also provides some facilities that permit definition of 'default' handlers, i.e. handlers that are invoked only if no higher-level handler resolves the signalled condition.

In a sense the MPS mechanism represents the "culmination" of the work in sequential program exception mechanisms, since little has been done to extend its applicability. If we review our list of issues (section 2.1), we find that it addresses most of them quite well. Exceptions are explicitly defined. The unwind mechanism helps preserve abstractions by guaranteeing that no executing context will be unceremoniously aborted by an exception. The mechanism meshes well with other language facilities and seems easily transferable to most conventional procedural languages, with acceptable cost. Why, then, is it not ideal for exception handling?

[6] The exception handling proposals of MPS have found their way into the MESA language [Geschke 77].

[7] We should note that, contrary to the time-sequence implied by the organization of this chapter, the BLISS signal-enable construct was actually derived from the MPS mechanism with deliberate simplifications to streamline implementation.

Perhaps the most serious shortcoming of the MPS mechanism is its failure to accommodate shared abstractions. When an exception is raised in MPS, the only possible source of handlers is the dynamic execution nest, i.e. the process's stack. This stack is searched in LIFO order until a handler is located that will 'accept' the exceptional condition. Hence there is a built-in assumption that the only contexts in which handlers can (should) be found occur in functions that have been initiated but not completed. Yet we can identify contexts in which the exception might be intelligently handled but which happen not to appear in the current call stack.[8] In essence, exception transmission in MPS and all the above mechanisms is forced to follow the "calls" hierarchy [Parnas 74], yet there is good reason to suppose that other hierarchies are appropriate as well. Indeed, it is the contention of this thesis that the "uses" relation is central to exception handling, and that the "calls" relation has been (mistakenly) applied instead because the two frequently coincide. When sharing of abstractions occurs, the relations diverge and the former becomes more desirable for exception processing. We will discuss the issue of sharing further in section 3.1.3.

2.2.8 Some Proposed Sequential Mechanisms

All of the exception handling mechanisms discussed above have been implemented in at least one system. We now consider briefly three mechanisms that, like this thesis, are proposals. Where implementations of these proposed facilities exist, they are confined to experimental systems and are not present in generally available systems or languages.

2.2.8.1 Parnas's Proposal

Parnas proposes a mechanism [Parnas 72c, Parnas 76] for error handling as part of his formal method of module specification [Parnas 72b]. His technique is to invoke externally defined error subroutines, which he views as analogous to hardware 'traps'. Thus each function of a module includes as part of its specifications a list of names of subroutines (which the user of the module must

[8] A specific example must await later terminology - see section 3.1.3.

provide) and defines the conditions under which each such routine will be called. This approach addresses several of our exception handling issues (section 2.1) nicely. Exceptions are explicitly and formally specified and are closely allied with levels of abstraction. Because errors (Parnas also uses the term "undesired events") are handled by subroutines, control may return to the detecting module after error processing has completed - a desirable property. Propagation of exceptions across levels is explicit, forcing the preservation of abstractions at each level. The mechanism is clearly language-independent and, because of its formal specification, likely to be verifiable as well. Parnas also recognizes the value of the "uses" hierarchy in preference to the "calls" hierarchy. Indeed, many of his suggestions are in line with the mechanism we propose in chapter 4.

Where, then, does Parnas's proposal fall down? Primarily in the absence of *specifics*. *While* advocating the "uses" hierarchy for exception transmission, Parnas gives no details about the binding of handler to exception, which crucially assumes a definition of the "uses" relation.[9] He also fails to specify important details of control *flow*, e.g. *how the choice is made*, after handler completion, whether to resume the signaller, abort the signaller, or retry the (failing) operation. It is, therefore, impossible to assess fully the merits of Parnas's proposal, and we must regard his exception handling mechanism as *incomplete*.[10]

2.2.8.2 Goodenough's Proposal

Goodenough [Goodenough 75] proposes a facility that in many ways closely parallels the MPS mechanism discussed above. Conditions are explicitly declared language entities and are raised by one of several special statements. These statements define the resumption requirements of the signalling context and are designed to be checkable at compile-time. Goodenough also predefines several special-purpose conditions for signalling loop termination, default handler invocation,

[9] His notation, however, suggests that at most one handler is bound to each exception, a restriction we strongly oppose. See sections 3.1.3 and 4.6.

[10] Wasserman [Wasserman 77] has extended the ideas of Parnas along lines somewhat *similar* to our 'flow' conditions (chapter 4), but relies solely on the "calls" hierarchy.

unwind, and other exceptions. Unfortunately, the interaction of these many conditions is difficult to understand, although their intended purpose is to simplify control flow and enhance clarity. We believe that Goodenough's mechanism, while properly limiting implicit propagation of conditions, fails to impose adequate constraints on the actions that handlers may take, particularly with respect to control flow.[11] Because it follows the MPS approach of using the dynamic context stack as the sole source of handlers, Goodenough's proposal suffers many of the same disadvantages with respect to shared data structures. In addition, Goodenough does not permit the passing of parameters to handlers (essential in our view), though he recognizes the desirability of doing so and acknowledges the deficiency of his mechanism in this respect.

2.2.8.3 Recovery Blocks

One other proposed mechanism should be noted, though its goals are rather different from those of the mechanisms previously discussed. The 'recovery block' approach of Horning et. al. [Horning 74, Randell 75] permits the specification of an 'acceptance test' for a body of program text. If the test succeeds, well and good, but if it fails, an 'alternate block' is invoked and, upon its completion, the acceptance test is performed again. This cycle continues until either the test succeeds or all alternate blocks have failed, in which case the 'failure' condition is propagated to the enclosing recovery block scope. One key point must be observed, however: alternate blocks are entered with the program state identical to that which existed at <u>entry</u> to the primary (or previous alternate) block. Thus all record of a failing block's execution is obliterated. (Efficient means for doing so are discussed in [Horning 74].) We see, therefore, that alternate blocks are not 'handlers' in the sense of MPS or BLISS, since the program state that led to the failure of the acceptance test has been lost. It is clear, then, that recovery blocks are not really intended as an exception handling mechanism. Indeed, the proponents of this facility are primarily concerned with capturing 'residual software errors' (bugs), not with handling unusual conditions that the software may legitimately detect.

[11] Specifically, handlers may have the option of aborting the signalling context - an unnecessary and dangerous freedom in our view. (See section 4.9.)

2.3 *Parallel Program Mechanisms*

We now turn our attention to existing structures for handling exceptional conditions in a parallel program. PL/I has already been mentioned as one of the few languages that attempts to combine sequential and parallel condition handling. We shall see that parallel and sequential program mechanisms have historically appeared fundamentally different, and that no real attempt has yet been made to combine them.

We should recall the remarks of chapter 1 about the differences between normal interprocess communication and exceptional conditions. The distinction, once again, is only a matter of degree. In a highly parallel system with only occasional interprocess communication, *every* such transaction might be regarded as signalling an unusual event. On the other hand, communication may be frequent with only a small subset of the interactions reflecting unusual events. In addition, the interactions, though frequent, may have little or no relationship to the computation(s) in progress at the instant of communication. Because of these vague distinctions, identical mechanisms have generally been used for both normal and exceptional interprocess communication, and our mechanism will be equally applicable as well.[12] Accordingly, we survey the major communication mechanisms in use, concentrating on their behavior as exceptional condition mechanisms.

2.3.1 Polling

Polling is perhaps the simplest form of interprocess communication. Processes using polling share a data structure in which they set status indicators from time to time. Status enquiries are made by explicit tests of the indicators, and synchronization is achieved by busy waiting. Polling can be used in any system in which (conceptually) parallel tasks and shared data are supported and thus is easy

[12] This again raises the issue of a single, unified view of control flow. We postpone discussion of this issue until section 9.2.7.

to implement, though frequently expensive to use. The objections to polling as an exceptional condition mechanism are analogous to those mentioned above in sections 2.2.1 and 2.2.2. No explicit structure is present, except that provided by other language mechanisms, so the identification of conditions, signallers, and handlers is purely *ad hoc*. The same holds for the techniques used to pass parameters between signaller and handler and to resume execution (conditionally) upon termination of the handler. Furthermore, the implicit access by multiple processes to the shared data structure enormously complicates understanding and verification of the behavior of the entire program. Finally, unless the processes possess and share knowledge about the underlying scheduling policies of the system, they will almost certainly waste processor time using busy waiting to achieve synchronization of signaller and handler. In short, polling is a seriously flawed exceptional condition handling technique.

2.3.2 Interrupts

Polling depends on the eventual inspection of *shared* data structures to initiate changes in control flow. An alternate method is to force the change by means of an interrupt, generated at the time an exceptional condition is detected. This is the traditional method used in hardware; a peripheral device notifies a processor of an exceptional condition (e.g. I/O completion) by causing a forced branch to a predetermined location. This differs crucially from the forced branch of section 2.2.2 in that the processor state at the time of the interrupt is saved as a side effect of the branching action. In addition, some of that state is replaced with new values taken from an area of memory associated with the interrupt location. When the interrupt handler has completed its work, the saved state may be reloaded and the interrupted program resumed.

Interrupts have two evident advantages over polling. First, the handlers are clearly identifiable and localized. Second, the need to test continually for the presence of the exceptional condition is eliminated, substantially reducing overhead cost. However, several problems remain. The shared data structure that complicates verification is still necessary, since parameters are not usually passed

with the interrupt. Also, the possibility of arbitrary interruption is intolerable to most programs; critical sections usually exist during which a crucial property (invariant) assumed by the interrupt handler does not hold. This leads to the notion of temporarily disabling, or masking, interrupts. In this way a program can prevent interruptions until it explicitly enables them. But then what happens if two interrupts arrive during a masked period? Are they queued separately or are they merged into one? If the latter (most common in hardware), how does the handler eventually discover that fact? And if the former, where are the queues constructed and how are they managed? Is it possible to mask each interrupt individually or are they grouped into classes which are masked or enabled as a whole? If the latter, is the grouping static or dynamic? These design issues and others have been extensively studied, since interrupts have been present in nearly every computer designed in the last 15 years. Some are appropriate for an exceptional condition handling mechanism with the properties we seek, others are not. We will close our discussion of interrupts by examining one useful extension to the basic notion.

Interrupts in their simplest form cause the flow of control to be transferred to a particular physical location, without any regard to the executing process. It is frequently more useful to be able to force a transfer of control to a particular virtual location within a specific process. That is, the interrupt is sent to a process (which is presumably interested in handling it), not to a processor. This notion is often available in software, but rarely in hardware, because of the absence (in general) of the notion of 'process' at the hardware level. However, it is obvious that an interrupt of this style is a much truer form of interprocess communication than the basic one previously discussed. In fact, it leads naturally into message-based communication.

2.3.3 Message Systems

Message systems dispense with the notion of (physical) processors entirely, and view communication as occurring between either processes or intermediate objects called *mailboxes*. Message system facilities are quite diverse

[Brinch Hansen 71, Wulf 78], and actually span a wide space extending from generalized polling to generalized interrupts. However, there are two important aspects of exceptional condition handling embodied in nearly all message systems: explicit parameter passing from signaller to handler, and implicit queuing (instead of merging) of distinct signals. By restricting the flow of data between processes to the well-defined channels of the message-passing mechanism, we remove many of the barriers to verification. If we use a message system in which the handler blocks its execution until awakened by the arrival of a message (i.e. it is not interrupted when a message arrives, but must explicitly ask the system for any outstanding messages), we further localize for verification purposes the interprocess communication. Retaining the identity of individual arriving messages also helps us prove that the processes synchronize in particular ways. In short, a message system appears to have the most suitable collection of properties (among existing mechanisms) for a parallel exceptional condition handling facility.

Some difficulties remain, however. Many message systems provide only unidirectional message flow, with no facilities for an implicit reply upon completion of the handler's (receiver's) actions. Thus the notion of resumption of the signaller must be built explicitly with a second 'return path' to the signaller. As we have seen, such unenforced explicit structures are harder to understand and verify than mechanisms which embody resumption in their semantics. More seriously, message systems are traditionally software structures supplied by the operating system or some subsystem. Thus they are not generally available for use at the hardware level or within the system itself, where much of the parallelism is likely to be. A different technique must be used within the system, introducing an undesirable non-uniformity in the exceptional condition structure.

2.4 Summary

As the preceding sections show, existing mechanisms for handling exceptional conditions in sequential and parallel programs differ considerably. We observe substantial overlap between exception handling and general communication, and some of the existing mechanisms cater more directly to one than to the other. The

mechanism proposed in chapter 4 respects that overlap while recognizing that certain concessions must be made if the goals of section 1.3 are to be attained. Now that we have both those goals and previous exception handling attempts in mind, we can proceed to a synthesis that will yield our proposed mechanism.

Part II
A New Mechanism

3
Elaboration of the Goals

In chapter 1 we presented the goals of this thesis. They can be restated in a single sentence. We seek to define a program structuring facility that permits the uniform handling of exceptional conditions, is amenable to rigorous verification, and naturally accommodates a variety of practical problems with reasonable efficiency. We have seen how existing mechanisms fail to satisfy these goals and thus have acquired some understanding of the undesirable properties embodied in these facilities. In this chapter we establish the groundwork for the presentation of a new mechanism that meets the stated goals.

Throughout this chapter the discussion will be at a general and informal level, postponing details until subsequent chapters. We cannot supply specific definitions for such crucial terms as "context", "module", "function", and "exception" until we have available to us the structure of a particular language. Chapter 4 will provide us with that structure; until then, we must content ourselves with intuitive notions and somewhat vague terminology.

3.1 *Uniformity*

The desire to exhibit a single, uniformly applicable mechanism derives from our attitudes about program structuring. We prefer to build programming systems from a small number of primitive concepts and to obtain expressive power from their interactions. Modern design precepts stress the organization of programming systems around a few elementary constructs whose semantics are carefully chosen and precisely defined. These primitives then permit us to build composite structures whose semantics are equally precisely defined [van Wijngaarden 75]. We are moving away from the "patchwork quilt" approach to language definition, which produces unmanageable programming systems assembled from disparate parts whose interfaces are jagged and frequently clash [IBM 70]. The mechanism of this thesis is intended to be one primitive semantic element of a modern programming system and may be impossible to stitch into the quilt of some existing languages.

PRECEDING PAGE BLANK-NOT FILMED

3.1.1 Choice of Language

We were careful in the preceding paragraph to use the term "programming systems" because we are deliberately proposing a mechanism that extends, in many ways, directly to the hardware level. As was stated in section 1.3.2, the implementation may vary with the system level, but this does not concern us. Conceptual uniformity is our goal, with identical semantics regardless of implementation level and technique. However, the primary application of this mechanism will be within programming languages (though not necessarily application-level programming languages), and therefore the bulk of the presentation in this thesis will be expressed in programming language terms. To do so, of course, we will need a language consistent with the precepts mentioned above. Several languages would be adequate for our purposes; the fundamental characteristic we require is *encapsulation*. This principle, also known as *information hiding* [Parnas 72a] or *necessary access* ("need to know"), states that programs receive only the minimum amount of information necessary to perform their specified functions. Founded on this principle is the program structuring methodology known as *modular decomposition* [Parnas 72d], which groups a data structure and the functions that manipulate it into a single entity called a *module*. The behavior of the functions is precisely specified in terms of abstract operations on the data structure; no information about the implementation of these operations or the representation of the manipulated structure is ever provided to a user of a module. Because we are concerned with the semantic integrity of our exceptional condition mechanism, we want to embed it in programming languages that stress abstract semantics rather than representational issues. Languages that prevent the spread of representation information (and thereby permit considerable flexibility in the choice of implementation) are thus well-suited to our purposes.

Given a choice of several acceptable languages, we see no reason to construct a hypothetical one expressly for the purposes of this thesis. We will use Alphard [Wulf 76a] for several reasons:

1) it supports the notion of encapsulation directly (through forms),

2) it addresses the issue of verification (see 3.2, below),

3) it is not yet completely specified, hence our proposal may be of practical value to its continuing development.

This last reason has two distinct facets. By casting our proposal in the context of an incomplete language, we may suppress or alter details that are not of central importance to our mechanism. At the same time, we may specify certain properties for language elements that are as yet undefined (e.g. parallelism). In both ways we hope to contribute to the process of refining a new language's specifications without doing violence to any of its essential notions.

3.1.2 Propagating Exceptions

A basic reason for the existence of exceptional conditions is the realization that the program context in which an unusual event occurs or is detected may not be the best context in which to *process it*. Consequently, this unusual condition is given a name, an *exception*, and the knowledge that it has been detected passes from the context that made the discovery to the context(s) that can process it. Generally, the point of detection and the point(s) of processing reside in distinct modules, so that the act of "raising the exception" causes information to flow between modules. In view of the comments of the preceding section, we should require that this information be consistent with the abstractions involved; that is, the detecting module should express the exceptional condition in terms of the abstraction the module defines.

It may not be possible for the recipient of an exception to process the condition completely. Often, the occurrence of the exception may have a serious effect on its behavior, forcing it to raise an exception as well. In accordance with the principle of maintaining abstractions, this second exception must be expressed in terms of the recipient's abstraction; it cannot simply restate the condition raised by the original detecting module. Thus, as a condition crosses an "abstraction boundary" (i.e. a module), it must be transformed to reflect the external semantics

and specifications of the module it is leaving. In this way, representational information remains captive within a module.

We can see an obvious analogy in the well-known problem of providing intelligible run-time diagnostics for a high-level programming language. Hardware-detected exceptions will be expressed in terms of addresses (the abstraction provided by the hardware), which generally have no meaning to the high-level language user. It is the responsibility of the language "module", which implements (through a compiler and run-time system) the abstraction inherent in the language semantics, to transform such addresses into the control and data structures of the program being executed. Most conventional systems transform only a subset of the exceptions the hardware can raise, and handle the remainder by printing a cryptic message (in terms of the abstraction supplied by the hardware) and terminating execution. Should the programmer be unfortunate enough to receive such a message (e.g. "Fatal Run-time Error: Memory Parity Error at 472106"), he cannot assign any meaning to it, since it is not expressed in terms of the abstraction (the language) he perceives. Equally cryptic messages can arise from software-detected exceptions as well;[1] in both cases an abstraction has not been maintained.

This example illustrates two errors in the design of the language module: its failure to express an exception in terms of the proper abstraction, and its unilateral decision to terminate execution rather than propagate a condition. We have just seen how useless an exception expressed in improper terms can be, but we should understand that the failure to raise an exception at all is just as serious.

Suppose that our hypothetical programmer has implemented a data-base system using the given high-level language. Because he is concerned with high reliability in his system, he has gone to considerable lengths to organize his data structures to permit recovery from various failures. However, he depends upon the language to provide him with exceptions for all conditions that it cannot handle internally. In the context of our previous example, we will assume that if the data-

[1] E.g. "Negative argument to LOG routine" triggered by the source-level expression X↑(1./3.), where "↑" denotes exponentiation and X is negative.

base system is informed (by an exception notification) that a particular datum has been destroyed, it may be capable of reconstructing that datum either from redundant data or recomputation. If it can do so, the fact that a datum was (temporarily) destroyed will be completely hidden within the data-base system and no exception will be transmitted to the user of that system. Should the data-base system be incapable of repairing the damaged item, it will probably report to its user some exception in terms of the abstraction the user sees (e.g. "Transaction for March 15, 1974, inadvertently destroyed"). All of this clearly desirable behavior depends crucially on the language module reflecting all exceptions to its user (the data-base system) in terms that the system recognizes.[2]

3.1.3 Associating Detection and Processing of Exceptions

We have already emphasized that an exception will generally be processed outside the module that raises it. A module that raises an exception will therefore not possess the knowledge to determine what program context(s) should receive notification of the exception; indeed, it will not even be able to name such possible contexts. The exceptional condition mechanism is therefore responsible for transmitting the exception to the proper context(s). Furthermore, the module raising the exception may need to synchronize with the context(s) processing the exception in order to determine when and if the difficulty has been resolved. Obviously, the mechanism must assume responsibility here as well, since the detecting module cannot explicitly name or enumerate the contexts with which it must synchronize. The means by which the exception mechanism performs these vital services are described in chapter 4; we concern ourselves here with several implications of our chosen methodology on the properties of this mechanism.

Informally, we can say that the contexts that require notification when an exception arises in a given abstraction are precisely the users of that abstraction. A careful definition of "users of an abstraction" depends on specific language

[2] Consider the havoc that the language module may wreak if it aborts execution of the data-base system instead of raising an exception. Crucial files of the data-base may be left in an inconsistent state, potentially causing further erroneous behavior when the data-base system is later restarted.

constructs, but we can give a characterization that depends on methodological views alone. We consider a context to be using an abstraction (or module) if it possesses the ability to invoke the functions provided by that abstraction. This definition is not quite vacuous; it permits a context to be using a module even though that context may not be currently executing. "Use" in this sense thus has both a static and a dynamic component: a context may be an executing program that can invoke the given module or it may be a data structure that is implemented in terms of that module. The former is a context defined by control flow, the latter is one defined by data flow, and of course the two are interrelated. In more careful terms, a context is using a module if it possesses a *capability* [Lampson 69] for that module. Thus, in order for the exception mechanism to notify the appropriate contexts of an exception raised by module M, it must be able to locate all capabilities for module M.[3] This does not imply that it will necessarily reflect the exception to all contexts holding such a capability, but merely that all such contexts will initially be considered eligible.

It is this notion of "possessing a capability" that defines, *for our purposes*, the "uses" relation mentioned in section 2.2.7. Several contexts are sharing an abstraction if each possesses a capability for it. We establish the "uses" relation, not the "calls" relation, as the basis for exception transmission by our mechanism. To see the difference in a specific example, consider the following situation (which reappears, thinly disguised, at several points in subsequent chapters).

Suppose we have a storage allocator that provides an abstraction called a "pool". This pool is shared by a number of users who may or may not be aware of each other's existence. Suppose that the allocator defines an exception, "pool-low", that it raises when insufficient resources are available to satisfy an allocation request. We might reasonably expect each user to release some storage when the pool becomes depleted, or at least, to *attempt* to do so by appropriate compaction of its private data structures. Thus, all users should have an opportunity, at least, to eliminate the pool-low condition when it is detected. Yet only one of the users will have performed the 'allocate' call that led to the detection. We see, therefore,

[3] In a sense, we are generalizing "object-oriented exception handling" (see [Goodenough 75], p. 685.) to "user-of-object-oriented" exception handling.

that the "calls" relation and the "uses" relation do not coincide in the presence of shared abstractions, and that the "uses" relation is intuitively more appropriate, at least in this case, for exception transmission. In later sections, particularly in chapter 7, we will acquire additional practical support for this view.

3.1.4 The Role of the Exception Mechanism

From the preceding discussion, it appears that the exception mechanism must supply a collection of services that permit modules to communicate the detection and processing of exceptions among themselves. Further, these services must not jeopardize (and indeed should encourage preservation of) the abstraction each of the modules provides. Let us consider, then, some desirable properties of the mechanism.

Modules that must operate correctly in spite of exceptional behavior in the abstractions they use should be designed with the principle of "mutual suspicion" [Schroeder 72] clearly in mind. Simply stated, this principle asserts that interacting modules validate (in terms of abstract specifications and insofar as is possible) the transactions between them. Normally, this principle is applied at function invocation, when one module explicitly calls another. Yet the same notion applies in exception handling. The module raising an exception is communicating with contexts that are expected to perform certain actions on their own behalfs and certain actions on behalf of the detecting module. The detecting module wants assurances that the contexts receiving the exception cannot prevent it from resuming execution; that is, it expects the mechanism to limit arbitrary transfers of control during exception processing. At the same time, contexts that are eligible to receive notification of an exception want assurances that they will indeed receive control if the exception is raised. We conclude that the mechanism must respect the mutually suspicious design of interacting modules and must not compromise that design at the exception handling interface.

The mechanism must not constrain the form of the abstractions a module designer chooses to define, as long as they are consistent with the general

methodological principle of information hiding. Whether a module provides a computational or structural abstraction is a matter for the designer; he should not be forced to choose a particular one because of the facilities provided by the exception mechanism. From this principle we infer that the mechanism should support with equal convenience exceptions related to flow of data and control.

We expect to justify the properties of the mechanism in terms of the information hiding methodology. We also expect the mechanism to fit neatly into the embedding language and to interact reasonably with other language properties. In particular, the mechanism must not require the language to provide specific facilities (e.g. synchronization) for the mechanism alone, but it must function predictably in the presence of such facilities. Parallel processing, shared data structures, synchronization, and protection are all language (system) notions that inevitably interact with exceptional condition handling, and we must demonstrate that the mechanism does indeed respond reasonably to all of these facilities.

The issues mentioned thus far in this chapter all concern the *requirements* imposed on the exceptional condition mechanism by the assumption that it is to be embedded in a language supporting encapsulation as a fundamental design principle. The generalities of this section will be translated in chapter 4 into specifics in the context of Alphard, and the reader will then be able to judge whether the goal of defining a uniformly applicable exception mechanism has indeed been realized.

3.2 *Verifiability*

Of the three primary goals established in section 1.3, verification has perhaps the most significant effect on the structure of the exception mechanism. The ability to verify programs in a particular language is enhanced by strong statements about global properties of the language, e.g. statements limiting the possible side-effects of out-of-line code. When we seek to extend that language with an exception handling mechanism, we must be careful not to weaken or invalidate any of these global assertions. In practice this means establishing strict rules for access to variables and transfers of control. It is often the case that two distinct

designs will supply essentially the same operational facility, yet one will be easier to verify than the other. That is, for the purposes of present-day proof technology, it may be possible to characterize formally the semantics of one mechanism more easily than the other. In the course of designing the mechanism to be presented in chapter 4, such situations have frequently arisen. We examine in this section the implications of the verification requirement on the mechanism; in chapter 6 we pursue the details of verification in the Alphard context.

3.2.1 Using Existing Language Properties

Because effective verification, with current technology, relies heavily on complete knowledge of the interactions of language facilities, we must approach our verification requirement in the context of a specific language, Alphard. We weave our mechanism into the language by borrowing the semantics of existing pieces of the language and modifying them as little as possible. In some cases this produces a design noticeably different from one that might evolve in the absence of a verification requirement, but as long as adequate function results, the approach succeeds.

Specifically, we borrow heavily from the notion of functions. Since conditions are detected and processed in separate modules, we cannot generally expect to have both detecting and handling code available simultaneously at verification time. When we verify a module, we expect to have only a characterization (in terms of predicates) of the semantics of the (external) functions and abstractions it employs. We use those predicates as we construct verification conditions for the invocations of those functions. If the externally-defined abstractions raise exceptions, it seems only natural to expect a characterization of the semantics of those conditions as well.

If we recall (from section 3.1.4) that the code to process an exception is required to return control to the module (function) detecting the exception, then we see a natural similarity between such code and a function body. The essential difference is that the "caller" (i.e. the module detecting the exception) defines

what the "callee" (i.e. the code processing the exception) may assume at entry and must ensure at exit. As long as those characterizing predicates are available in both modules, however, the verification paradigm is quite similar to that for normal functions. Indeed, the proof rules developed in chapter 6 are, in essence, slightly altered versions of conventional proof rules for functions.

Thus, the verification requirement has led us to choose an exception mechanism that resembles closely the normal function invocation mechanism. In so doing we are able to use variants of existing proof rules without developing entirely new constructs or proof methodologies. Although we may sacrifice some efficiency in the operational behavior of the mechanism, we obtain instead an enhanced language that remains verifiable. We believe the exchange is a profitable one.[4]

3.2.2 Predicates

Without upstaging the development in chapters 4 and 6, we present briefly the consequences of choosing a function-like view of exception processing. As stated above, we will assume, when verifying a module that handles an exception, the availability of predicates that characterize the action to be taken. Specifically, we will require that exceptions be named explicitly, and that every exception have two associated predicates. These predicates are pre- and post-conditions on the abstraction that incurred the exception; that is, they define, respectively, its state before and after the handling code has been executed. From the viewpoint of the code that handles the exception, the pre-condition may be assumed at entry, and the post-condition must be satisfied at exit.

Naturally, the pre- and post- conditions supplied by the module defining the abstraction can only express properties of that abstraction. The code that handles the exception will have pre- and post-conditions as well, but these predicates will

[4] In section 5.2 we examine ways that the implementation can overcome some potential losses in efficiency incurred by this exchange. By providing adequate "syntactic sugar", the language may encourage a user to express program behavior in such a way that the compiler/run-time system can discover "optimizations" that restore lost efficiency.

include information about other abstractions in use besides the one that raised the exception. The proof rules of chapter 6 relate these predicates to each other to establish the formal semantics of the exception transmission and processing.

Two side issues should be briefly noted. First, we are using a proof methodology that derives directly from Hoare's axiomatic method. [Hoare 69] We do so because Alphard does and we need to be consistent with the methodology of the specific embedding language. This does not imply, however, that other techniques are unsuitable. Indeed, there may be more elegant formulations that make verifications involving exception handling more convenient. We do not explore that possibility here, satisfying ourselves only that a particular well-known proof methodology can be used to verify programs using this mechanism.

Second, we should also note that the specific form of the predicates mentioned above depends on other language facilities. If the language provides machinery for explicit state characterization of abstractions, the predicates will very likely be expressed in terms of such states. Otherwise, the state information will be implicitly encoded in the data structures used to implement the abstraction, and the predicates will reflect those encodings. There is disagreement among researchers regarding the value of explicit state characterizations for verification purposes; we do not pursue that issue here.

3.3 *Adequacy*

Since we claim that the mechanism of chapter 4 is adequate for the solution of practical problems, we need a collection of real-world exception handling problems to use as examples. These examples should demonstrate the scope of the mechanism's power and should include some problems that cannot be solved reasonably with the existing mechanisms discussed in chapter 2. Let us briefly examine several such problems, all of which are treated in detail in chapter 7.[5]

[5] Chapter 7 demonstrates the mechanism's ability to accommodate traditional exception handling problems (e.g. arithmetic overflow, incorrect parameter to subroutine) as well.

3.3.1 Memory Data Error

General-purpose computing systems rarely handle memory errors in an intelligent manner. Typically, a memory error within a user program causes that program to be stopped and receive a message. A memory error within the operating system generally induces a crash. (We have already discussed the undesirability of such behavior in section 3.1.2.) Special-purpose systems have developed particularized mechanisms for recovering from memory errors, but such mechanisms rely on the structure of the application, not general principles of program organization and information flow. We want the facilities of the exception mechanism to encourage coverage of detected errors (both hardware and software). Perhaps the lack of an adequate mechanism has fostered the defeatist approach that is embodied in most present-day systems.

3.3.2 Resource Allocation Failure

The issues related to exception handling in a resource allocator (e.g. a primary memory allocator) have been briefly mentioned in section 2.2.6. We may summarize the discussion by identifying three distinct exceptions that an allocator might reasonably raise and observing their different properties.

- In the course of satisfying an allocation request, the allocator may detect that its resources, while adequate to meet the current request, are running low. Such a condition should probably be propagated to all contexts where resources might conceivably be made available to the allocator. Yet there is no reason to suspend the current allocation request while the condition is being handled; the two actions are logically independent and should proceed (conceptually, at least) in parallel.

- The allocator may find it impossible to satisfy a resource request and may raise a condition expressing its urgent need for storage. Naturally, the request will remain pending while this exception is being processed.

- The allocator, even after raising the previous condition, may not possess adequate resources to satisfy the request. In this case, it must raise an exception signifying its inability to meet the requestor's demands.

We note the differences in synchronization and communication requirements among these conditions and their handlers. Doubtless we could postulate other plausible conditions that the allocator could raise, conditions with still different processing needs. Existing exception handling mechanisms do not provide the facilities required by these various situations; we will demonstrate in chapter 7 that the proposed mechanism solves the allocator problem completely and naturally.

3.3.3 I/O Completion

The preceding examples embody situations in which the exception being raised is not crucial to normal program behavior. That is, if the condition never occurred, the program would execute just as well (perhaps better!) and terminate normally. However, we can identify conditions that are essential for normal termination, e.g. the knowledge that a previously initiated i/o operation has completed. By proposing that our exception mechanism provide natural means of handling conditions that arise during normal processing, we are overlapping somewhat into the area of general inter-program communication. We do not claim that our mechanism is adequate for general communication, but we observe that it is sufficiently robust to handle situations in the gray area between exceptions and communication. We consider an example from this domain.

Suppose we have a collection of processes that maintain a display of various quantities computed from data supplied by external sensors. Such a collection might be part of an aircraft control system. Naturally, some quantities (e.g. altitude) are more crucial than others (e.g. cabin temperature) and consequently are to be recomputed as soon as new sensor data arrives. We may choose, for example, to assign these high priority computations to a specific subset of the processes, allowing them to participate in the less important computations only when they have fulfilled their primary responsibilities. Should such processes be performing low-priority calculations when new sensor data arrives, we will want them to switch back to high-priority calculations, using the new data.

This system could be (and probably has been) programmed without any exception handling mechanism, but would undoubtedly use knowledge specific to the task to construct the necessary communication and synchronization facilities. The exception handling mechanism of chapter 4 is sufficiently general to permit this problem to be solved without additional special features; in particular, without assuming any explicit process synchronization mechanisms. By solving this problem (and related ones involving relatively infrequent synchronization), the mechanism demonstrates its adequacy in specialized areas of inter-program communication.

3.4 *Practicality*

Practicality in a language facility implies the existence of an efficient implementation. However, efficiency can be an elusive notion. By removing a single instruction from a program, we undoubtedly increase its speed, but the degree of that increase depends crucially upon the frequency with which the instruction had been executed. Furthermore, the frequency in executions per second is not interesting; rather, we want to determine frequency of execution relative to the entire program. That is, we measure the speed-up (or should measure it) in terms of the original execution time. This is perhaps an elementary principle, but one often ignored by programmers, who have been known to "optimize" sections of a program that were responsible for only a small percentage of the total execution cycles consumed.

How is this observation relevant to exception handling? We are concerned that the cost of setting up handlers for conditions that occur infrequently (in a relative sense) not be excessive. In particular, for an exception mechanism to be practical, it must be competitive with other language facilities that might otherwise be used to achieve the same effect. If our mechanism is too expensive to use, programmers will find other means to achieve the desired effect, probably decreasing program clarity at the same time.[6] Thus we measure the efficiency of our exception mechanism as its cost of use relative to other language constructs.

The cost of a language feature may be subdivided into two components. One cost is incurred whenever the feature is used explicitly, such as dynamic allocation of variables or procedure invocation. The second cost is distributed over the implementation so as to lessen the first cost at the time the feature is explicitly invoked. The distributed cost will be incurred whether or not the feature is used. An example is the non-local goto in ALGOL-60, which when used incurs a considerable expense (compared to a local goto). The variable display must be "unwound" to restore the context at the destination of the goto. However, the very means used to maintain the display in order to handle a non-local goto represent a distributed cost of this facility, since additional processing is needed at block entry and exit. A distributed cost is normally the result of interacting language features rather than a single facility, as in this interaction of variable allocation and non-local transfer of control in ALGOL-60. We will want our exception mechanism to minimize the distributed costs of its implementation, hence we will carefully control its interactions with other language facilities.

When the language implementor is faced with alternative realizations of a language feature that distribute the costs differently, he tends to favor one or the other on the basis of anticipated use. Such a bias can occur in the design process as well. Though the exception handling mechanism of chapter 4 may be suitable, in principle, as a more general communication mechanism, its design has

[6] An example of such behavior occurs in the "real-world" use of PL/I. Procedures are an important program structuring device, yet the prohibitively high cost of procedure invocations in some implementations of PL/I has caused programmers to avoid using them. The resulting programs have long, multi-function procedures, which are difficult to understand and maintain.

been influenced substantially by its intended application to exceptions. Thus, in practice, it might prove more costly than is necessary or desirable in a communication mechanism. The design emphasizes the view that one is willing to pay a considerably greater cost to transmit an exception than to perform most "normal case" operations. Thus the specifications for the semantics of raising an exception will probably force a more costly implementation than that of, say, function invocation. At the same time, the design recognizes that the action of "enabling" a handler (i.e. providing a body of program text that, in a particular context, is willing to process a given exception if it arises) is itself a "normal case" operation, and thus the specifications of the mechanism permit this action to be implemented reasonably inexpensively.

As a final observation on the subject of implementation efficiency, we recall that the implementation of the exception mechanism need not be uniform at all levels of a system. The notion of "context" inside the operating system may permit a more efficient realization of the exception mechanism than does an arbitrary high-level programming language. The specifications of the general behavior of exceptions will be consistent; the language-specific details (concerning interaction with other facilities) will vary. It is also conceivable that the full generality of the mechanism may not be needed within a particular language (e.g. the ability to handle exceptions in an environment that supports parallel processing). In such a case, a consistent subset of the mechanism's facilities may be implemented, with a potential saving in execution cost. The relative cost of the mechanism will therefore vary depending upon the other facilities available in the environment; the design only ensures that, for a broad class of "reasonable" languages, the relative cost is acceptably low.

4
The Mechanism

This chapter concentrates on the specifics of the proposed exceptional condition handling mechanism. We will examine the functional semantics of the mechanism and relate them to the facilities of a particular programming language, but the exposition is nevertheless relatively language-independent. We postpone until chapter 7 examples that require specific syntax for their presentation.

4.1 Terminology

In preceding chapters we have used a variety of terms to suggest the different notions associated with exceptional condition handling. Before proceeding to a careful definition of our mechanism, we must agree on a particular set of names to identify the concepts we will discuss. Here is a list of these terms with their intended connotations:

Condition - We use this term to identify an exception, in the sense of section 1.1. Normally, conditions will have names suggestive of their interpretation, e.g. 'memory-data-error'. The problem of name conflict will be discussed later (section 4.4.1). Occasionally, the term *exception* will be used synonymously with *condition*.

Signaller - A program that raises a condition is called a *signaller*.[1] Generally, we will prefer to say that a given program "raises a condition" than to say that it is a signaller, but we will find it convenient to have a single word to contrast with 'handler' (see below). 'Raiser', though consistent, seems too awkward to bear frequent repetition.

[1] The term 'signaller' derives from the signal-enable mechanism of BLISS (see section 2.2.6).

Handler - A piece of program text that is intended to process a specific condition is called a *handler*. As we shall soon discover, handlers may appear in many places within a program, and the selection of the handler(s) that will actually receive notification of a given condition (when it is raised) is a crucial issue. A particular handler is associated (statically) with a single condition, although various language mechanisms may be employed to share program text among handlers.

Eligible - When we refer to a handler as *eligible*, we mean that, if the condition with which the handler is associated is raised at this instant, the handler will be among those considered for execution. Whether it is actually invoked depends upon the *selection policy* for the condition, a property we will cover in section 4.7.1. A handler that is not eligible cannot be invoked, regardless of selection policy. The precise circumstances that determine eligibility are discussed in section 4.6.

Context - Perhaps the most overworked term we shall need to use is *context*. For the purpose of exception handling, we shall normally apply the term *context* to the execution environment of a piece of program text that is under consideration. Thus, for example, we may refer to the "context in which the memory-data-error handler is invoked", i.e. the control operations and data accesses permitted when the specified handler executes. The terms "domain" and "environment" are roughly synonymous, but generally refer to a broader, more slowly changing notion (e.g. "protection domain") than "context", which includes the more local connotation of flow of control along a particular path. Our most frequent application of the term will be to the execution environment of an eligible handler.

A considerable number of additional terms will arise in the course of this chapter, but most of them have not suffered the extensive history of varied

application of those above. We will borrow freely from Alphard terminology to avoid creating a vocabulary from scratch.

4.2 Relevant Language Notions

Throughout the remaining sections of this chapter, we will examine the detailed structure of the exception mechanism. Although the discussion will use Alphard where reference to language specifics is necessary, we can best demonstrate the language-independence of the mechanism by falling back on Alphard details as infrequently as possible. In this section we introduce some programming language notions that will appear frequently in the sequel and relate them to specific Alphard constructs.

We will use the term *module* to refer to the general unit of program structuring. A module provides an abstractly-defined service or implements an abstract concept, e.g. symbol table, stack, or rectangular matrix. Modules provide abstract specifications of their behavior and hide the implementation of that behavior. Thus we are directly borrowing the term 'module' from Parnas, as previously discussed in section 3.1.1. Modules are sometimes called types, type modules, forms, modes, clusters, protected subsystems, and other things, although these terms do not all carry precisely the same connotations. Depending upon the emphasis placed on issues such as protection, formal specification, verification, and hierarchical structure, one of these terms may be more appropriate in a particular discussion than others. We will skirt these semantic pitfalls by using *module* exclusively, except when we refer specifically to the Alphard construct, which is called a *form*.

Modules provide their services to their users through *functions*, in Parnas's terminology. We will adopt the term function, although we want to strip away the mathematical connotations that are inappropriate for programming. Functions are essentially subroutines[2] whose names are known outside (i.e. to the users of) a

[2] Even 'subroutine' has undesirable connotations. We do not preclude macros, out-of-line procedures, or synchronous interprocess communication.

module. Specific languages may provide the notions of "internal" and "external" subroutine; our functions correspond most closely to the latter. However, functions are more than just externally visible subroutines, for their behavior is precisely specified to the user in terms of the abstraction implemented by the module. Alphard also uses the term *function* for this notion, and thus our general and specific terminology coincide.

The concept of *sharing* is fundamental to the handling of exceptional conditions. Issues related to sharing arise commonly in discussions of operating system facilities, for example, whether or not two users may share a resource (a file, a page, a peripheral). Implicit in these discussions is the concept of *instance;* there is (say) a single notion 'file', but many instances of that notion, some of which may themselves be shared. Translating this into our previous terminology, a module provides an abstraction which may be instantiated many times. Each instance will be available (accessible) in one or more contexts, in the sense of section 4.1. For example, a file containing accounting data might be accessible only in contexts that exist within the module providing login/logout functions.

When we instantiate a module, we produce, at least conceptually, a copy of everything it contains. Thus all the functions of the module and all the internal data structures are duplicated according to a rule supplied by the module itself.[3] In implementation terms, copying a function generally does nothing more than supply the address of that function; that is, because the code is usually not modified, the implementation permits the textual body of the function to be shared. However, conceptually, each instance provides a distinct copy of the function text. A less trivial example is the internal data structure of a module, which frequently is physically copied for each instance of the module's abstraction. Thus every time a 'stack' module is instantiated, a separate top-of-stack pointer and stack segment are created. Distinct stacks naturally have distinct top-of-stack pointers. Note, however, that copying of data in a language that permits indirect references (so-called ref modes) may simply imply that distinct instances have distinct pointers to a single shared structure. That is, sharing arises from copying references to structures rather than the structures themselves.

[3] Thereafter, of course, separate instances are independent - changes in one do not induce corresponding changes in the others.

We should also recognize that the term *instance* may be applied to both data flow and control flow. When we instantiate a file, we create a copy of it (more precisely, we instantiate the module 'file'). When we instantiate a function, we invoke it. Thus if two distinct users simultaneously invoke the 'file-read' function, we refer to them as having distinct instances[4] of that function. In a sense, each user is executing his own "copy" of the function. Within this meaning of 'instance' as applied to control flow, there is no room for sharing. We may easily see what sharing of data instances means: multiple contexts possess the ability to access the same data instance. However, every separate locus of control at every instant of time resides within precisely one function instance (by definition), and thus the sharing of such instances is impossible. This peculiar asymmetry of data and control instances will be explored further in section 4.5.

Alphard has no specific terminology for the notions of instance and invocation. We note, however, that specific notation does exist to control the instantiation process for an individual form (the "rule" alluded to above). Within a form, each data element (i.e. the internal data structure of the form) is identified as unique (i.e. private) or shared. When the form is instantiated, copies are made of all unique data elements; shared elements are common to all instances of the form. We note in passing that certain kinds of sharing are not permitted by this method, although some not directly available may be derived by appropriate layering of forms and the use of ref variables. These details are not relevant to discussions of the exception mechanism, since its design supports all classes of sharing consistent with module structure of the kind previously described. The Alphard facilities meet that requirement.

4.3 A Gentle Introduction

We now have acquired the necessary background to discuss the details of the proposed exception handling mechanism. To aid in the understanding of the specifics presented in subsequent sections, we provide here a sketch of the main features of the mechanism, with forward references to the detailed text.

[4] "Activations" is perhaps a more familiar term.

Refer to figure 4.1. We define two modules, T1 and T2, with the former using the abstraction provided by the latter. That is, for every instance of T1, there is a distinct instance of T2, which is referenced inside the body of T1 by the name 's'. These semantics derive from the standard Alphard declaration on line 3.

```
1      module T1
2        begin
3          unique s:T2
           . . .
4          function f =
5            begin
               . . .
6              g(s)
               . . .
7            end [c: h(v) ]
           . . .
8        end
9      module T2
10       begin
11         condition c(v:integer)
           . . .
12         function g(t:T2)
13           begin
               . . .
14             raise c(17)
               . . .
15           end
           . . .
16       end
```

Figure 4.1: A Simple Example

Although this simple example in no way illustrates the general flexibility and power of the mechanism, by "walking through" it we can acquire a sense of the workings of the mechanism. Suppose, then, that within some instance of module T1,

function 'f' is executing. That is, some user has declared a T1 and is applying 'f' to it. Suppose that 'f' has invoked function 'g', in module T2. At the moment that we turn our attention to the example, 'g' is about to execute line 14.

An exceptional condition has been detected by 'g'. The declaration on line 11 *defines* the condition to have the name 'c' (details in section 4.4). This name is made available to the users of T2's abstraction (e.g. T1) in the same way that the names of the functions (e.g. 'g') are. The statement on line 14 *raises* the condition 'c', which causes the user of the abstraction to be notified.[5] This means some code is to be executed to *handle* the exceptional condition. We see this code on line 7, represented by 'h'. Of course, the detector of the exception may wish to pass additional information to the handler besides the condition name. Such a parameter is defined in the condition declaration (line 11) to be an integer 'v'. Its actual value at the time of detection (line 14) is '17', and it is transmitted with the condition name to the handler 'h' (line 7), which we presume is a procedure appearing elsewhere within T1.

Immediately, a wealth of questions arises. How was the code on line 7 determined to be the handler for this condition? (See section 4.6.) Could other handlers be invoked instead or as well? (Yes - see sections 4.6 and 4.7.) What information does 'h' have available to it and what is its execution environment? (See sections 4.5 and 4.8.) How does 'h' influence the subsequent behavior of 'g' and 'f'? (See sections 4.8 and 4.9.) These questions, and others, will all be addressed in the course of this chapter.

Returning to our example, we have passed the exception 'c' to the handler 'h', together with some parameter information. We (usually) view this notification as a subroutine call; that is, 'g' is temporarily suspended while 'h' executes. (See section 4.8.) 'h' performs whatever recovery actions it chooses (consistent, of course, with the semantics of 'f' and 'c'!) and terminates. Control then returns to 'g' immediately after line 14. (See section 4.9.)

[5] Just what constitutes "the user" and how does the notification occur? We'll address those issues in sections 4.6ff.

More questions arise. Must 'h' always return control to 'g'? (See sections 4.7 and 4.9.) Must 'g' always wait for 'h' to complete? (See section 4.7.) Can 'h' alter the data or control flow in 'f'? (See section 4.9.) How are the answers to these questions affected by sharing T2's abstraction among many separate instances? (See sections 4.5 and 4.7.) What about simultaneous parallel execution of 'g' by more than one process? Parallel executions of 'f'? (See sections 4.6 and 4.11.) We see that there are a considerable number of important issues to address in defining an exception handling mechanism. It is the intent of this example to raise these issues; it is the task of subsequent sections to handle them.

4.4 *Conditions*

What is a condition? That question has a philosophical component, which we have already examined (see section 1.1). In this section we concentrate on the fundamental aspects of defining conditions within a chosen programming language.

4.4.1 Condition Names

We declare conditions in much the same way that we declare other structuring elements. In Alphard we might write

```
condition no-space-left
```

thereby defining an exception. Such a declaration would appear in the "specifications" portion of an Alphard form, which contains information to be exported to the form's users. (In later sections of this chapter, we will extend this declaration to include additional information necessary for proper handling of exceptions.)

Conflicts between condition names in a particular form and names exported by other forms are handled in the same way that function name conflicts are resolved.

Where ambiguity can arise, the language must provide a "qualifying" notation to permit the user (programmer) to specify the intended interpretation of a symbol. The concept of name qualification to remove ambiguity is well understood and not really relevant to the definition of an exception mechanism. Suffice it to observe that whatever technique is used elsewhere in a language to resolve conflicts will serve for condition names as well.

4.4.2 Parameters

Frequently, a handler requires more information about a condition than simply its name. The signaller may wish to transmit additional information to a handler, and may do so by means of *parameters* associated with the condition. In this regard, the specification of a condition appears rather similar to that of a function; in Alphard we might write

```
condition illegal-symbol(bad-symbol:string)
```

This notation indicates that a handler for condition 'illegal-symbol' may expect a single parameter (of type 'string'[6]), which (presumably) is relevant to the handling of the condition. As we shall see later, a handler may be viewed as a subroutine that is invoked when a particular condition is raised. Thus our use of function-like notation to describe parameter lists is deliberately suggestive.

4.4.3 Conditions and Instances

When an exception is raised by a function within a particular module, the user of the module must know whether the exception refers to a condition that now exists within the module's internal structure or applies only to the function invocation itself. The distinction is most important when multiple users share the same abstraction (module). As an example, consider a module that implements a file abstraction. The 'file-write' function specifies two conditions (among others) that it

[6] I.e. defined by form 'string'.

When we define an exception in Alphard, we do not explicitly specify its class as a structure or flow condition.[9] Instead, we name the instance(s) on which the condition may be raised. For example, the 'file-write' function above might begin

```
function file-write(f:file)
   raises file-inconsistent on f
   raises file-read-only on file-write
   . . .
```

This notation indicates that 'file-inconsistent' is raised on a structural entity (a file) and is therefore a structure class condition, while 'file-read-only' is raised on a function instance (file-write) and is therefore a flow class condition. If more than one file might be subject to the 'file-inconsistent' condition, we might write

```
function file-compare(f,g:file)
   raises file-inconsistent on f,g
   . . .
```

The raises clause is always associated with a function specification and is naturally exported to the users of the function (actually, to the users of the form). The condition declaration often appears outside the function declaration and groups information about the condition (e.g. its parameter list) in a single place. This is convenient when the same condition may be raised by more than one function within a form. Additional illustrations may be found in the examples of chapter 7.

4.5 *Handler Definition*

We now introduce the concept of a *handler* for a condition. In section 4.1 we presented an informal view of the actions performed by a handler; here, we discuss the particulars of handler definition and relate them to Alphard.

[9] We stress that these terms are for descriptive purposes only. They are not necessary for the careful definition of the exception mechanism, but are introduced only for convenience in discussing its properties and its implementation.

PRECEDING PAGE BLANK-NOT FILMED

To discuss a handler by itself is impossible; we must relate it to the condition it purports to process and the surrounding program text. In Alphard, we might write

```
S [C: H]
```

where 'S' is a program element (e.g. a statement or block), 'C' is a condition name, and 'H' is a handler. We say that H is a handler for condition C that is *associated* with S. It is important to stress that S might be a function invocation, a statement involving several operations, or an entire block complete with local declarations. In short, S may be any executable program unit from a primitive operation (function invocation) up to and including an entire function body. (Under certain important circumstances, S may be an entire module body, but more on this later.) We should also note that H is treated as a part of S for the purposes of identifier resolution; that is, in principle H has access to all symbols accessible inside S. Our notation is not ideal in this regard, since H does not appear in the correct lexical position for such access to occur as a result of the normal scope rules.

We impose some restrictions on the general form of H. First, it must not contain language constructs that might irrevokably transfer control outside its lexical scope. This restriction derives from the semantics we will require at handler termination time and will be discussed in section 4.9. Of course, H may perform function invocations, but it may not "permanently" transfer control outside its lexical boundaries.[10] H also may not attempt to return from the function (if any) in which it and S reside. Second, H is further restricted to simplify the task of verification, as will be discussed in chapter 6. However, these restrictions will not materially constrain the nature of the computations H may perform, only the syntactic form they may assume.

We recall from our previous discussion (section 4.4.3) that conditions and instances are closely allied. A condition is not raised *in vacuo*, but relative to a specific instance. A notation that emphasizes this important point is:

```
S [I.C: H]
```

[10] E.g. by a goto or exit construct.

Here we are explicitly relating the handler H to a condition C raised on instance I (for the duration of S). We say that H is *attached* to instance I. Although we will frequently omit I when it is unambiguously determined by the local context, its conceptual importance cannot be overstated.

The full justification for this fuss over instances and handler attachment will come in sections 4.6 and 4.7, but we can give a preview here. When a particular condition occurs (is raised), we will want to notify all contexts that might be interested in that condition. Since we raise a condition relative to an instance, we will need to know all contexts that have handlers attached for the given condition and the particular instance on which it was raised. We will claim that this set of handlers contains as a subset the handlers that should actually be invoked. Thus it is essential that we be comfortable with the notion of attaching handlers to instances. Let's consider some examples.

Returning to the file example of section 4.4.3, we may write the quasi-Alphard statement

```
file-write(accounts) [file-read-only: h]
```

Since 'file-read-only' is a 'flow' class condition, it refers to control instances (function calls). Thus in this case, handler 'h' for condition 'file-read-only' is *attached* to an instance of function file-write. At the same time, 'h' is *associated* with the program fragment 'file-write(accounts)'. Thus the evident *intent* of this statement is to invoke the 'file-write' function, and if it raises the 'file-read-only' condition, to handle it by the code body 'h'. The instance on which 'file-read-only' is raised is the invocation of 'file-write' - a control instance.

Now consider data instances. Suppose we write

```
file-write(accounts) [file-inconsistent: h]
```

Since file-inconsistent is a 'structure' class condition, we are attaching 'h' to a structural instance (a data structure), presumably 'accounts'. To avoid any confusion, we could use the ambiguity-resolution convention of Alphard and write

```
file-write(accounts) [accounts.file-inconsistent: h]
```

Here, the program context with which the handler is associated is identical to that of the preceding example - an invocation of 'file-write'. However, the condition 'file-inconsistent' is a structural one, so the instance to which 'h' *is attached is* not the invocation of 'file-write' but 'accounts', an instance of the 'file' abstraction.

Clearly, we may want to perform different actions to handle the 'file-inconsistent' condition (*raised on* 'accounts') at different times. Were this not the case, we would simply attach the handler 'h' to 'accounts' for the entire lifetime of the file. Instead, our notation permits us to attach 'h' to 'accounts' only for the duration of some context, in this example, a single function call. We can identify two notions of "lifetime" here: the lifetime of the instance to which the handler is attached, and the lifetime of the context with which it is associated. For the 'file-read-only' example above, the two lifetimes were identical; for the 'file-inconsistent' example, the latter was contained in the former (i.e. 'accounts' existed before and after the execution of 'file-write'). Conventional exception handling mechanisms have generally recognized only one of *these lifetimes as* important (usually the latter). Our mechanism explicitly permits variability in both dimensions.

Consider another example. In the Alphard program fragment

```
begin
    private x:integer
    file-read(old,x);
    file-write(new,x+1);
    file-write(transaction,x);
end [new.file-inconsistent: h1 | file-read-only: h2]
```

we assume that the second parameter to 'file-read' is a var and the second parameter to 'file-write' is a value.[11] The intent of the program fragment is to read a value 'x' from file 'old', write 'x+1' to file 'new', and write 'x' to file 'transaction'.

[11] A var parameter may be altered by the callee, a value parameter may not.

Note that, since 'file-read-only' is a flow class condition raised by file-write, it could conceivably be raised at two distinct points in the execution of this program unit. Thus there will be two instances of file-write to which 'h2' might apply. We interpret the above notation as a shorthand that attaches a separate copy of 'h2' to each relevant instance inside the program unit with which the handler is associated.[12] The situation with 'file-inconsistent' is rather different. This is a structure class condition, and handler 'h1' is evidently to be used only if the condition is raised on file 'new'. If 'file-inconsistent' were raised on either 'old' or 'transaction', this context would provide no handler to process the condition. Depending on the larger context in which this fragment appears, such a situation might constitute a programming oversight or a legitimate, deliberate structure.

Observe the relationships that exist between context and instance lifetimes. h1 is attached to 'new.file-inconsistent' only for the duration of this program unit, which is evidently a subinterval of the lifetime of 'new'. By contrast, 'file-write.file-read-only' has h2 attached for its entire lifetime; in fact, the notation suggests that h2 spans several lifetimes (separate instances). Let us examine this lifetime matter a little more carefully, and at the same time return to the question of handler placement in the source program text.

For flow class conditions, it is clearly impossible to change handlers during the lifetime of the instance to which they are attached. A control instance is nothing more than a function invocation, which is a primitive operation from the viewpoint of the context that performs it. Yet a data instance is a structure that normally exists across primitive operations. Consequently, a program may wish to vary the handler action for a structure class condition according to the context of execution at the moment the condition is raised. In particular, our example above shows that if 'file-inconsistent' is raised on the file 'new' in the given program context, handler 'h1' is to be used. Other contexts that manipulate 'new' may supply other handlers. Specifically, we may want to handle conditions over the entire lifetime of a data instance, just as we do (by definition) for control instances. If that lifetime falls completely within the execution lifetime of a function body, then our mechanism already provides a means of accommodating such handlers, e.g.

[12] This is a conceptual requirement only. Naturally, the implementation will share the body of 'h2' rather than create actual copies.

```
begin
   private tempfile:file
   . . .
end [tempfile.file-inconsistent: h]
```

However, if a structure's lifetime exceeds a single function body's execution lifetime, this method is inadequate. Such structures are of necessity declared in the "outer block" of a module.[13] For consistency, we allow a handler to be associated with an entire module body, thereby attaching it to an instance for the module's entire lifetime,[14] e.g.

```
form account
   begin
      private acct:file;
      . . .
   end [acct.file-inconsistent: h]
```

All of our preceding comments about symbols accessible within 'h' and the lexical form 'h' must assume still apply. Obviously, it makes no sense to attempt to specify at the module-body level a handler for a flow class condition;[15] only data structures can persist across function invocations.

A final comment about module-body level handlers. The eligibility rule (see below) will require that, within a module that references an instance, there always exist at least one attached handler for every condition that can be raised relative to that instance. We enforce this requirement by assuming that, if no handler

[13] For otherwise they would be declared within some function, and exist only for the duration of an invocation of that function.

[14] Strictly speaking, for the lifetime of the reference to the instance. Alternatively, we might associate choose to the handler body with the declaration of the instance, thereby emphasizing that the handler is attached for the duration of that declaration's relevance.

[15] Except in the sense of the shorthand notation previously explained. We allow such handlers at the module-body level for that reason alone.

exists for a given condition and instance, the programmer intended to write one with a null body (i.e. a handler that does nothing) at the module-body level. It will be convenient to make the same assumption when verifying the module (see chapter 6).

4.6 Handler Eligibility

When a condition is raised, we may wish to invoke more than one handler to process it. The choice of handlers to be invoked occurs in two stages: a set of *eligible* handlers is determined, then a subset of the eligible handlers determined by a *selection policy* is chosen. This subset will then receive the opportunity to process the condition. In section 4.7.1 we discuss selection policies; this section is devoted to the notion of eligibility. We first approach the subject informally, then proceed with more precise definitions.

Roughly speaking, we define a handler H to be eligible to handle condition C on instance I if it is associated with a program context that is able to access I. By 'access' we mean the ability to name the instance I. For flow class conditions, this definition is essentially trivial, since the only context that can 'access' a function invocation (control instance) is its caller.[16] Thus for flow class conditions, there can be at most one context in which an eligible handler may be found.

Structure class conditions are not as simple. Because of sharing, a single data structure may be referenced by several contexts. In most cases we have no *a priori* basis on which to distinguish these contexts, and we treat them all as eligible. (This statement will be qualified shortly.) This implies that a context may be considered eligible even though it did not invoke the function that raised the condition. Furthermore, if the system permits parallel execution, that context may be performing some other function at the instant that the condition is raised. We will examine the synchronization details in section 4.11; for now we need only be aware that multiple handlers may be eligible to receive the same (structural) condition.

[16] More precisely, the instance of the invoking function.

We observe that multiple eligible handlers may exist even if the programming system does not permit parallel execution. Suppose, for example, that we have two modules, A and B, that use a single storage allocation module, S.[17] Assume that S provides an abstraction called a 'storage pool', which is shared by all of its users (in particular, A and B). Assume further that S defines a structure class condition, 'pool-low', that is raised whenever its argument pool has inadequate free space to satisfy the requested allocation. Since we assume no parallel execution is possible, only one module can invoke 'allocate' at a time. Suppose A does. If 'allocate' raises 'pool-low', should we *a priori* deny B the opportunity to handle the condition and free some space? Certainly not. A and B share the storage pool provided by S, and our mechanism should and will consider both modules eligible to handle conditions raised by S. Sharing, not parallelism, produces multiple eligibility.

Let us now proceed to the details of eligibility determination. Since flow and structure class conditions may be treated analogously, we will focus our attention on the latter class, since the absence of sharing of control instances makes the former class less interesting.

4.6.1 The Eligibility Rule

In order to give a precise definition of eligibility, we must introduce some terminology. First, we need to tighten up our meaning of "context". Consider an executable program unit, that is, a well-formed piece of source code that forms a subpart of a function body (perhaps the entire body). If, at a given instant, that code is being executed, then an instance of the function body must exist. Indeed, a separate instance will exist for each distinct execution of the function, regardless of whether such executions are serial or parallel. Within each instance, there will be an instant at which our chosen program unit is entered and another at which it is exited. Between these two instants, the program unit is said to be *active* (with respect to the given function instance). In a parallel system, the same

[17] This example appeared earlier in section 3.1.3.

program unit may be active in many different function instances at once.[18]

We next restrict our attention to active program units that are relevant to a particular condition and instance.

Def. Suppose that condition C has been raised on instance I. Consider all active program units that have an associated handler for condition C attached to instance I. (Of course, some of these will be properly nested, but we'll deal with that in a minute.) We call this set of active program units the *contexts* in which C raised on I can be handled. We will say that a handler is *enabled* if either (1) it is associated with (an instance of) a module body in which I is referenced (named), or (2) it is associated with a context in which C can be raised on I. (For reasons indicated at the end of section 4.5, we always consider module-body level handlers to be enabled.)

We need one additional preliminary definition.

Def. Suppose H1 and H2 are distinct enabled handlers and S1 and S2 are respectively the program units with which they are associated. We say that H1 *masks* H2 if either (1) S2 is a module body within which S1 is lexically nested or (2) neither S1 nor S2 is a module body, but S1 is lexically nested in S2 and the contexts associated with H1 and H2 are subparts of the same

[18] We have carefully avoided using the term "process" because of our unwillingness to assign it a specific meaning in this discussion. "Process" has too many implementation-specific connotations. However, we note informally that (barring recursion) simultaneously occurring instances of the same function belong to separate processes. In the subsequent paragraphs we will establish definitions that, without resorting to the definition of "process", pinpoint within each process that references a data instance the contexts in which the signalled condition should be handled.

function instance.[19] An enabled handler H is said to be *masked* if there exists some other enabled handler G such that G masks H.

We can give a precise definition of eligibility.

Def. A handler is *eligible* (to process a condition on a given instance) if it is enabled and not masked.

To see what this means more concretely, we will extend an earlier example - our storage allocator S and its users A and B. Let 'usea' and 'useb' be functions in modules A and B, respectively. Sketches of these modules, written in pseudo-Alphard, appear in figures 4.2 and 4.3.[20] 'Allocate' is assumed to attempt to assign a 'block' of 'i' storage units from the specified pool 'p'. If it cannot do so, it raises 'pool-low' on 'p' and, after all eligible handlers have completed, tries again.[21] If the second attempt fails, it returns 'nil'.[22] The 'release' function is assumed not to raise any exceptional conditions.

First, let us assume that no parallelism is permitted. (The two modules as coded are prepared to handle potential parallelism, which we discuss in the next

[19] A nearly equivalent, and perhaps slightly more intuitive, way to state (2) is: neither S1 nor S2 is a module body, both S1 and S2 belong to the same module, and S1 is dynamically nested within S2. Unfortunately, this definition appeals implicitly to the notion of "process". It also complicates verification.

[20] A word of caution is in order: these examples are somewhat simplified. The declarations of 'p' on lines 3 and 16 are intended to denote the same structure, although in strict Alphard they would denote different ones. We are also omitting certain details about handler interactions that have not yet been explained. See section 4.9 or, better still, ignore the details for now.

[21] Again, we ignore until section 4.7 the mechanism that makes this possible.

[22] Once again, we must simplify for illustrative purposes. Normally, 'allocate' would raise a second condition here, e.g. 'pool-empty', thereby permitting its users to avoid the cumbersome tests for 'nil'. To avoid confusion at this point and because we would otherwise have to presume material from section 4.9, we choose the traditional expedient of an unusual return value to denote an exception.

```
 1    form A =
 2      begin
 3        shared p:pool
 4        private k,l:hairylist
 5        function usea(i:int) =
 6          begin
 7            private z:block
 8            z←allocate(p,i)
 9            if z=nil then return fi
10            < fill in 'z' >
11            enter(l,z)
12          end [pool-low: squeeze(l)]
13      end [pool-low: (squeeze(l); squeeze(k))]

  Note:  'squeeze' is defined in form 'hairylist'.
```

Figure 4.2: A Simple Use of the 'Pool' Abstraction

paragraph. The "no-parallelism" assumption merely causes the handler on line 29 to be superfluous.) Suppose that 'pool-low' is raised as a result of the call to 'allocate' on line 8. By our assumption, the only function instance from which 'p' can be accessed is 'usea'. Consequently, the enabled handlers are those on lines 12, 13, and 31. However, by clause (1) of the definition of masking (above), the handler on line 12 masks the one on line 13. Thus the eligible handlers are on lines 12 and 31. If, however, 'pool-low' were raised by the call on line 23, the enabled handlers would be on lines 13, 23, 30, and 31. By clause (1) of the definition of masking, line 30 masks line 31; by clause (2), line 23 masks line 30. Thus in this case the eligible handlers would be on lines 13 and 23.

A more interesting situation arises when we allow parallelism. Suppose two processes independently invoke 'usea' and 'useb'. Specifically, at the instant that process 1 calls 'usea', process 2 is executing line 28 in 'useb'. If 'pool-low' is raised as a result of 'usea' calling 'allocate', the enabled handlers will be those on

```
14      form B =
15        begin
16          shared p:pool
17          private m,n:hairylist
18          function useb(i:int) =
19            begin
20              private z1,z2:block
21              z1←allocate(p,i)
22              if z1=nil then return fi
23              z2←allocate(p,i+3) [pool-low: release(z1); z1←nil]
24              if z1=nil
25                then return
26                else if z2=nil then release(z1); return fi
27                fi
28              < fill in 'z1' and 'z2' >
29              begin enter(m,z1); enter(m,z2) end [pool-low:  ]
30            end [pool-low: squeeze(m)]
31        end [pool-low: (squeeze(m); squeeze(n))]
```

Figure 4.3: A More Involved Use of the 'Pool' Abstraction

lines 12, 30, and 31, and hence the eligible handlers will be on lines 12 and 30. Note that, since 'useb' is being executed concurrently at the instant 'pool-low' is raised, the handler on line 30 masks the one on line 31. Consider what happens if process 2 has just passed the begin on line 29 at the instant 'pool-low' is raised. Now line 29 contains an enabled handler as well, and by clause (2) of the definition it masks line 30, leaving 12 and 29 as the eligible handlers. The purpose of coding a null handler body on line 29 is to take advantage of the masking to disable temporarily the unwanted behavior of line 30. The full power of this masking effect cannot be appreciated until we have discussed handler semantics in sections 4.8 and 4.9. We will return to this example in chapter 7.

4.6.2 Another View of 'Contexts'

Because the notion of eligibility rests at the heart of our exception handling mechanism, we must have a clear picture of its purpose. The definitions of the previous section identify for us the contexts that will be considered eligible to process a given condition raised on a given instance. Perhaps by recasting these definitions in operating system terms we can reinforce the intended ideas. We will use the terminology of Hydra as defined in [Cohen 75].

A context, in Hydra, is an object. Objects are used to group capabilities for related objects. We may view an object in which a particular capability resides as defining the environment (i.e. the capabilities) needed when the given capability is manipulated in a specific way. Stated slightly differently, the object defines a (not necessarily unique) usage environment with respect to the particular capability. (Of course, the object serves this same purpose for each of the capabilities it contains.)

For a specific instance I (which is an object), we may consider the set S of objects that contain a capability for I. Assuming that every object in S contains a body of code to handle an exception C raised on I, we can say that S is the set of contexts in which C (raised on I) can be handled.[23] This corresponds to the definition of "context" in the preceding section.

At a given instant, S may contain objects of type LNS (that is, objects corresponding to function activations). Indeed, for flow class conditions (i.e. conditions raised on I when I is itself an LNS), S has precisely one member, which is of type LNS. We make no special provisions regarding LNSs or non-LNSs - a context may be an object of arbitrary type. This is in basic contrast with other exception handling mechanisms.

[23] Strictly speaking, we should limit S to the set of objects that contain a capability for I but do not form a part of the protected subsystem that implements the type to which I belongs. We clearly do not want to include the implementing module (environment) among the "using" modules (environments).

The eligible handlers set is determined (at a given instant) by identifying within each member of S the body of code that is prepared to handle condition C. Clearly, for each context, the eligible handler may change with time. The eligibility rule given in the preceding section selects a single eligible handler from the enabled handlers *within a single context* (in S). It does so by the usual lexical scope rules. However, we should realize that the fundamental principle, for the purposes of our mechanism, is the determination of the set S. Within each context in S, the policy used to determine the eligible handler (at a given instant) is of secondary interest. T'ere are really two levels of selection here: the environments (contexts, protection domains) that use I, and the code bodies within these environments that will actually process C. The first level is central to our mechanism, the second is not. Within the latter we have chosen the conventional scope rules for familiarity and convenience, but the same exception transmission principle could easily be applied to another choice. It is important to recognize these distinct levels, even though they do not appear explicitly in the (rather "homogeneous") definitions of the previous section.

4.7 Raising Conditions

We now turn our attention to the semantics at the site where a condition is raised. We want to establish what control the signaller has over the invocation of handlers and what the signaller is entitled to assume after such invocation. Let us examine these issues in turn.

4.7.1 Selection Policies

By definition, a condition can only be raised within the module in which it is defined (i.e. in which the condition declaration appears). In Alphard we might raise 'file-inconsistent' by the executable statement

```
raise file-inconsistent
```

We assume that the declaration of this condition (see section 4.4.3) appears in the specifications part of a form, i.e. the part of the module that describes the abstraction to its user. The raise statement above would appear in one or more of the functions of the same form. For example, consider the following skeleton function:

```
body file-write(f:file)
   begin
      if not consistent(f)
         then raise file-inconsistent fi;
      ...
   end
```

'Consistent' is a predicate that tests the concrete representation of a file to determine if it is indeed consistent. Again, we assume that the above code appears within form 'file'.

Now let us consider the choice of handlers to be invoked. In preceding sections we defined the notion of eligibility but suggested that it might not always be desirable to invoke all eligible handlers when a condition is raised. We establish the principle that an eligible handler should be invoked only when the condition for which it is eligible exists. At first glance this seems a vacuous statement, but a simple example will demonstrate otherwise. Consider the 'pool-low' condition in the example of the previous section. In general, more than one eligible handler exists for this condition. Suppose that 'pool-low' is raised. It must be that insufficient storage exists in the (shared) pool 'p' to satisfy some allocation request. Assume for the moment that the (eligible) handlers are invoked one-at-a-time in an unspecified order. It is quite possible that the first handler will release adequate storage to satisfy the pending request. Why, then, invoke the remaining handlers? In actuality, the 'pool-low' condition has ceased to exist after termination of the first handler, and thus by the above principle, we should not invoke another eligible handler, even though many more may exist.

This example in addition illustrates a particular *selection policy*. This policy would cause handlers to be successively invoked as long as the given condition

persists. We will specify the precise semantics of this policy shortly; first, however, we observe that other reasonable policies exist. For example, consider the following variant of the 'pool-low' condition. Suppose that the allocation module, when a request could not be satisfied, wished to recover as much storage as possible within the pool. Indeed, it might wish to do so as soon as the available space dropped below a certain threshold, not necessarily when a request could not be honored. Such a condition, say 'pool-under-threshold', might be broadcast to all eligible handlers; by definition, 'pool-under-threshold' would be said to persist from the detection of the threshold-crossing until all eligible handlers have completed execution.

Another facet of selection policies is the degree of parallelism they permit among the eligible handlers and the signaller. Must the signaller wait for all selected handlers to terminate? Do the selected handlers execute sequentially or in parallel? The answers to these important questions significantly influence the structure of the handlers and the signaller, as well as the precise semantics of the condition. We could easily define a large number of policies by varying the subset of eligible handlers selected and the extent of parallel processing among them. Indeed, we naturally ask how we can characterize the space of selection policies.

The principles of modular decomposition and information hiding enable us to formulate properties that selection policies should have. From these properties we will seek an appropriate way to define acceptable policies within our exception handling mechanism.

4.7.1.1 Properties for Selection Policies

Because handlers appear, by definition, in modules other than the signalling one, the signaller has no knowledge of their individual actions. It thus makes little sense to provide a mechanism that permits explicit selection from the set of eligible handlers. The signaller has no a *priori* knowledge about handlers; it cannot, therefore, determine in advance those that should be invoked. For the same reason, the signaller cannot specify the order in which selected handlers are to be invoked. Hence a necessary property of an acceptable selection policy is that it cannot postulate the existence of a *priori* information that distinguishes individual eligible handlers.

Even in the absence of handler-specific information, we can still formulate policies that require only a subset of the eligible handlers to be invoked. Such policies rely on the state of the signaller before handler selection. An example of such a policy is that described above for the 'pool-low' condition, which uses the state of the storage pool to determine if additional handlers should be invoked. A consequence of the "anonymous handler" property, therefore, is that a (proper) subset of the eligible handlers may be chosen by the selection policy, but the determination of that subset must follow solely from information available within the signalling module.

The second property a selection policy should possess affirms the "principle of condition persistence", mentioned above. That is, the selection policy should not permit a handler to be invoked when the condition for which it is eligible does not exist. Frequently, the selection policy may come close to violating this property, as for the 'pool-low' condition above. 'Pool-low' and 'pool-under-threshold' differ only slightly, but the selection policies and subsequent behavior of the signaller differ substantially in the two cases. In fact, we might view the semantics of 'pool-under-threshold' to have been derived from those for 'pool-low' in such a way as to satisfy the dictates of the "condition persistence" principle.

A corollary of this principle is that all handlers selected by a policy must have been (at least) invoked before execution of the raise statement completes. If this were not so, then a handler could be invoked when no raise on its condition is in progress - a direct violation of "condition persistence".[24]

The final property a selection policy must exhibit is perhaps less obvious than the preceding two. We require that the policy be associated with the condition, not with an occurrence of a raise statement naming that condition. Thus all raise statements for a condition will necessarily use the same selection policy. The reader may find this restriction difficult to accept, since it seems plausible that different selection policies might be useful under different circumstances. Experience suggests, however, that such situations, if they arise at all, really involve two subtly different conditions (e.g. 'pool-under-threshold' and 'pool-low').

[24] Some unpleasant consequences of relaxing this principle are mentioned briefly in section 4.11.

Furthermore, in such cases handler action is generally slightly different under the two policies. This would imply providing the handler with the knowledge of which policy was in force at its invocation. Such knowledge would violate the principle of information hiding,[25] and would complicate verification. It is not impossible to construct verification conditions in the absence of this restriction, but, unless the policies are very similar, the proofs are likely to become much more difficult.

4.7.1.2 Specific Selection Policies

With these properties in mind, let us define some selection policies. We do not claim to exhaust the space of acceptable policies, but those defined here will be adequate to illustrate the intent of the above rules. We anticipate that an implementation of our proposed mechanism would supply (some of) these policies and perhaps others, but would not necessarily supply a set of primitives that permit the definition of new ones.[26] We will identify each policy by a keyword for convenient future reference.

Our first policy is called *broadcast-and-wait*. Under this policy, all eligible handlers are invoked in parallel.[27] When all handlers thus initiated have completed, execution resumes immediately following the raise statement. Thus, while all eligible handlers execute (conceptually, at least) in parallel, the signaller is suspended until they complete.

An obvious variation of this policy, simply called *broadcast*, relaxes the

[25] Since selection policies are bound only to use information available within the signalling module, there is no reason (other than synchronization requirements - see below) for handlers to know what policy led to their invocation.

[26] No such primitives are examined or proposed in this thesis. The identification of suitable primitives for the definition of selection policies is an open problem.

[27] We should note that parallel execution is specified by this policy and others to permit systems that support parallelism to apply it here. Systems that are defined to be sequential (not parallel systems running on a sequential machine) have no need for such a policy, since they can achieve the same effect by using sequential-conditional (see below) with the predicate identically false.

completion requirement. All eligible handlers are initiated in parallel, but the signaller does not wait for any of them to complete. It merely continues executing following the raise statement. Thus all handlers and the signaller execute in parallel.

The last policy we will define here is *sequential-conditional*. (This policy was used implicitly in the 'storage pool' example above.) Associated with the raise statement is a predicate. The predicate is evaluated and, if true, execution continues following the raise statement. If the predicate is false, an eligible handler is selected and initiated. When it completes, the predicate is again evaluated. (We insure that each time a different handler is selected.) The raise statement terminates when either the predicate becomes true or the set of eligible handlers is exhausted.[28]

We close this discussion of selection policies by relating them to condition classes. Since the class (structure or flow) of a condition determines the set of eligible handlers, the effect of the selection policy, which chooses from that set, obviously depends on the condition class. We note that the above policies are applicable to either class, although for the 'flow' class certain degeneracies occur. Since there can be at most one eligible handler for a 'flow' class condition, broadcast-and-wait and sequential-conditional are (in practice) equivalent.

This 'special-case' nature of flow class conditions reveals an important aspect of selection policies: they are generally interesting for structural conditions only. The possibility of multiple eligible handlers arises from shared (data) instances; it is for shared structures that this exception handling mechanism has been designed. Some previous mechanisms try to operate in environments in which (by definition) only one eligible handler exists, and such degenerate handler sets are demonstrably inadequate for shared data structures.

[28] It might be useful to alter this algorithm slightly. In practice, it is useless to test the predicate before selecting the first handler, since a true predicate would cause the raise statement to do nothing. However, it is useful to know after the raise that the predicate is false if and only if all eligible handlers have been invoked. Altering the algorithm to test after invocation prevents this conclusion. It also affects verification - see section 6.4.2.

4.7.2 The Raise Statement

In terms of control flow semantics, the raise statement acts much like a function call. Although the signaller does not know how many handlers will be invoked, it does know that, at some point,[29] control will return to the statement immediately following the raise. The signaller knows that control may leave its module for some period of time, but nothing any handler can do (short of looping indefinitely) can prevent control from returning. (See section 4.9.) The moment of return is determined by the selection policy.

For an ordinary function call, the caller is entitled to assume a certain predicate holds after the function is completed. That predicate is specified by the implementor of the abstraction, namely, the module in which the function is defined. When raising exceptions, the situation is much the same: the signaller is entitled to assume that a certain *predicate holds after the raise* statement has been executed. Once again the predicate is supplied by the implementor of the abstraction, but in this case, the implementor is the same module that contains the raise, by definition. Because the signalling module supplies the predicate, it compels the handlers to make that predicate true.[30] We will examine the consequences of this form of semantic specification in chapter 6.

It is important to note that few, if any, other exception handling mechanisms require control to return to the signaller upon handler termination. Some permit it, but do not enforce it. In our view, handlers are invoked (in most cases) to process an unusual condition that cannot be handled entirely within the signalling module. They have an obligation, expressed by the predicate, to the signaller, and the signaller will assume, upon completion of the raise, that the obligation has been satisfied. If handlers were able to terminate prematurely the execution of the signaller, the abstractions of the signalling module might not be maintained. The

[29] Assuming all handlers eventually terminate.

[30] Of course, it must not set impossible requirements, or no other modules will use its services!

signalling and handling modules should be viewed as mutually suspicious subsystems [Schroeder 72]; neither should be able to influence adversely the execution of the other. For this reason, our exception mechanism requires that handlers return control to the signaller. From the signaller's point of view, a handler is nothing more than an unnamed function.

A final detail concerning the raise statement should be mentioned. It may be necessary to indicate on what instance the condition is being raised. If, for example, there is a function 'compare-file' that compares the contents of two files passed as parameters, it will be necessary to specify which of the two parameters is intended if 'file-inconsistent' must be raised. We might write

```
raise file-1.file-inconsistent
```

to avoid ambiguity. Such qualifications are necessary only when a function manipulates two or more (data) instances of the same 'type' during a single call. No problem can arise in specification of flow class conditions, since the current executing function instance is the only possible relevant one.

4.8 Handler Invocation Semantics

We come at last to the semantics of the handlers. We can view the task of a handler as consisting of two parts: performing recovery actions on behalf of the signaller, and altering the local context at the handler site to reflect the occurrence of the condition. (For any particular condition, either of these actions may be null.) Thus a handler requires information from the signaller, which it receives through the condition name and parameters (see section 4.4), and from the associated program context, which it receives by the normal scope rules of the language.

Recall from section 4.6 that an eligible handler always has an associated program unit, which may be executing at the instant the handler is invoked. If so, we require that the execution of that program unit be suspended until the handler

terminates. That is, the handler *interrupts* its associated program context when selected for execution; it does not run asynchronously. This is a crucial distinction in light of the handler's responsibility to alter local context to reflect its recovery actions. If the handler executed asynchronously with its associated context, complicated and costly synchronization would be needed to prevent undesirable interference in the manipulation of local data.

It is important to understand that the associated context is interrupted immediately. Even though the context may be performing some function whose implementation exists outside the scope of the current module, interruption is immediate and causes that function's execution to be suspended. Failure to do so can result in a deadlock.[31] We can restate the rule in a less formal, though perhaps more intuitive way by saying that the "process" in which the program context is executing is interrupted, and the handler then executes within that process with access to the local variables of the context with which it is associated. The handler does not have access to the variables of the immediately interrupted context unless it is, by coincidence, the context with which the handler is associated. (Recall, however, that we prefer to avoid definitions that rely on the notion of "process", since the embedding language may not provide any such concept at the user level.)

We observe that reliance upon processes leads to difficulties when we consider handlers that are associated with entire module bodies. Recall from section 4.6 that such handlers are eligible only when no function within the module body has the condition enabled (relative, of course, to some particular instance). In informal terms, there is no activity within that instance - the data structure is not being manipulated. We cannot necessarily identify a "process" that should be interrupted in order to invoke the eligible handler, yet that in no way should prevent its execution. The data to which the handler should have access is well-defined by its lexical placement; the "control environment" is unspecified. Probably the

[31] Consider, for example, the situation in which the only eligible handler is associated with the context that invoked the function that raised the condition. If we wait to invoke the handler until control returns to its level of abstraction (i.e. the level at which its associated program context executes), we have a deadlock, assuming the selection policy involves waiting for handler completion.

implementation will have to find a "null" process in which to execute the handler, but at the level of functional behavior this detail does not concern or interest us. We merely specify the context in which the handler must execute and allow the implementation to realize that specification any way it chooses.

4.9 *Handler Termination Semantics*

To complete our exposition of the exception handling mechanism, we examine the semantics that apply when a handler completes its execution. There are two obvious aspects to consider: the effect on the signaller and the effect on the interrupted context.

Under some selection policies, the signaller is unable to continue execution until the handlers have completed. The signaller will need to assume, in most such cases, that it is impossible for its execution to be aborted, for the abstraction it maintains is very likely to be in an inconsistent state at the instant a condition is raised. As intimated in the preceding section, we require that a handler be incapable of preventing the resumption of its signaller. This is in sharp contrast to most existing exception handling mechanisms, which either permit or prescribe termination of the signaller. This abortion of the signaller may be satisfactory for certain conditions, but is inadequate for situations in which the signaller wishes to act upon the results of the handler (e.g. a storage allocator that, after raising 'pool-low', wants to determine if it now has adequate resources to satisfy a pending request). We reject it as a general rule. Under the proposed mechanism, the signaller's execution always continues immediately following the raise statement.[32]

Although a handler cannot alter the flow of control in its signaller, it can change the local flow of control within its associated context. This means that the handler can cause control to resume at a point other than the point of interruption. The precise nature of the control flow changes permitted is, of course, language-dependent, but for the purposes of this section, we assume only that the handler has the option of causing control to leave the associated context. (In effect, this

[32] Under some circumstances, this may be too cumbersome. See discussion in section 5.2.

gives us a kind of "local abort".) We recall, however, that the handler may not transfer control outside its own scope (see section 4.5). The actual transfer of control occurs not when the handler terminates, but *when control returns to its associated context at the point of interruption.* Stated another way: the handler, upon completion, "posts" the location at which execution of its associated context is to resume. Normally (and by default), the posted location is the point of interruption, but if the handler so specifies, it may be the textual end of the program unit.[33] The transfer to the posted location occurs when control returns to the interrupted context.

```
function useb(i:int) =
l:begin
    private z1,z2:block
    z1←allocate(p,i)
    if z1=nil then return fi
    z2←allocate(p,i+3) [pool-low: release(z1) → leave l]
    < fill in 'z1' and 'z2' >
    begin enter(m,z1); enter(m,z2) end [pool-low:  ]
  end [pool-low: squeeze(m)]
```

Figure 4.4: Improved Version of 'useb'

We can see the utility of local alteration of control flow for conditions that represent the permanent failure of some action. Returning to our allocator example again, we can see the convenience of being able to leave the body of function 'useb' (see figure 4.3) when 'pool-low' is raised as a result of the call to 'allocate' on line 23. We could then eliminate much of the troublesome checking for 'nil' that clutters the main-line code. Extending the notation of section 4.5, we might write

S [C: H → T]

[33] Or perhaps some enclosing context, depending on the branching constructs permitted by the language.

where T specifies the change of control flow that is to be posted when H terminates.[34] We could then recode 'useb' as shown in figure 4.4.[35]

4.10 *An Informal Recapitulation*

In this chapter we have expounded the details of a proposed exceptional condition handling mechanism. Because of our stated intention to do so outside the confines of a particular language's syntax and semantics, the presentation may have seemed vague and the factual detail elusive. Chapters 5 to 8 attempt to justify the definitions of this chapter by re-examining them in the light of the goals stated in chapter 1. For convenient reference and to help the reader understand the motivation for the elaborate definitions of this chapter, we restate here the important notions embodied in the proposed mechanism.

* The exception mechanism presumes an embedding language that explicitly supports the principle of information hiding, which separates knowledge of functional behavior (abstraction) from internal workings (representation or implementation).

* Conditions are defined by the implementor of an abstraction, and their names and abstract meanings are made available with the names and abstract meanings of the functions provided by a module.

* Handlers are executable program units (normally procedures) supplied by the user of an abstraction to process exceptions (conditions).

* The notion that binds handlers to conditions is the instance. Only

[34] This notation is not always convenient. See appendix A.

[35] Note that this improved version could still be improved by addition of other conditions, as observed in section 4.6.

those contexts that use a particular instance can supply handlers for the conditions that may arise on the instance. Put slightly differently: the implementor of an abstraction detects and raises conditions on its instances; the users of the instances supply the handlers to respond to those conditions.

* Notions like "process" and "call-stack" are secondary, implementation-specific concepts that do not figure directly in the transmission of exceptional conditions. We phrase our definitions and organize our thinking with the view that there exist instances of abstractions and users of those instances. That dependency relation is foremost; relations like "A calls B" are secondary.

* Conditions are defined to apply to either control instances (function invocations) or data instances (data structures). This 'class' of the condition determines what is meant by a 'user' of the instance on which the condition is raised.

* When a condition is raised, two criteria determine the handlers that will be invoked to process it: the eligibility rule and the selection policy. The eligibility rule defines the contexts within modules using the instance that will be considered eligible to have their associated handlers invoked. The selection policy defines the subset of the eligible handlers that will actually be invoked.

* Since the eligibility rule allows for dynamic changes in control flow, the set of eligible handlers for a given condition and instance varies with the state of the modules supplying the handlers (the users). The selection policy permits dynamic state changes in the implementation of the instance (resulting from handler execution) to affect the actual invocation of handlers. Thus both user and implementor can influence the exception handling process, but only in well-defined ways.

* Handlers cannot alter the flow of control within the signaller, but they are able to force local branching within their associated contexts. Functions that raise conditions may therefore assume that their execution cannot be terminated by external means. Local changes in control flow initiated by handlers actually simplify the structure of their associated contexts by eliminating the need for explicit polling for exceptions.

4.11 A Postscript: Synchronization Esoterica

We have presented the details of the exception handling mechanism without paying particular attention to questions of synchronization, even though at several points we stated that certain program units would execute in parallel. Two synchronization issues arise: one pertains to synchronization within the exception handling mechanism itself, the other involves the interaction between the exception mechanism and the synchronization facilities of the embedding language. We will consider the former here; we postpone the latter until chapter 5. Because the material of this section is relevant only to implementation, it has been separated from the discussion of the functional behavior of the exception mechanism.

The first observation about synchronization behavior applies to maintenance of the eligible handler set for a given instance and condition. This set is defined at the moment that the condition is raised on the instance, but the implementation will maintain it over time so that it is available at the instant the condition is raised. The definitions require that the eligible handlers set remain unchanged while the raise statement is being executed, but since the set is determined by the instantaneous control points of asynchronously executing processes, it is liable to change at any moment. It is necessary, therefore, to prevent interfering changes by ensuring certain mutual exclusion. Specifically, any attempt to alter or examine an eligible handlers set must be performed in a critical section protected by a mutual-exclusion semaphore (or similar synchronization construct). Such accesses occur in three places: entry to and exit from a program unit with which a handler for

the condition and instance is associated, and during a raise of the given condition on the given instance. By inserting critical sections at these points, the required semantics of the eligible handlers set are preserved.[36]

A second observation applies to selection policies that do not stipulate waiting for the handlers to complete. At first glance, we might worry that, under such a policy, control could "back out" of the signaller and into a context interrupted by a handler. The confusion would be considerable if this could happen, but a little care in the definition of the selection policy eliminates any potential problem. Note that the 'broadcast' policy of section 4.7.1 specifies that the raise statement completes *after* all eligible handlers have been initiated. Any eligible handler in the call stack of the process performing the raise will *not* only have been initiated but also will have completed before control can pass to the statement following the raise. Recall that the process that executes the raise will be interrupted immediately by a handler execution if the context with which the handler is associated happens to be in that process. This is in accordance with the rule stated in section 4.8. Thus no special machinery is needed to handle this apparent possibility of conflicting control flow; we need only exercise care in the definition of selection policies, and ensure that they exhibit the properties stipulated in section 4.7.1.

[36] It should be noted that the precise nature of the implementation may streamline these critical sections considerably. Without going into the details, we note that primitive "spin-locks" may be used, if appropriate, instead of full-fledged semaphores, and that techniques exist to build semaphores that are very inexpensive to use if blocking is rare. Also, certain implementations may permit the removal of an eligible handler (e.g. at block exit) from the set to be accomplished in a single instruction. If the underlying hardware ensures mutual exclusion at either the instruction or the memory cycle level, it may be possible to dispense with any software-implemented mutual exclusion. One can even postulate reasonable architectures in which addition of a handler to the set *might* be done indivisibly at the hardware (firmware) level.

Part III
Justification of the Mechanism

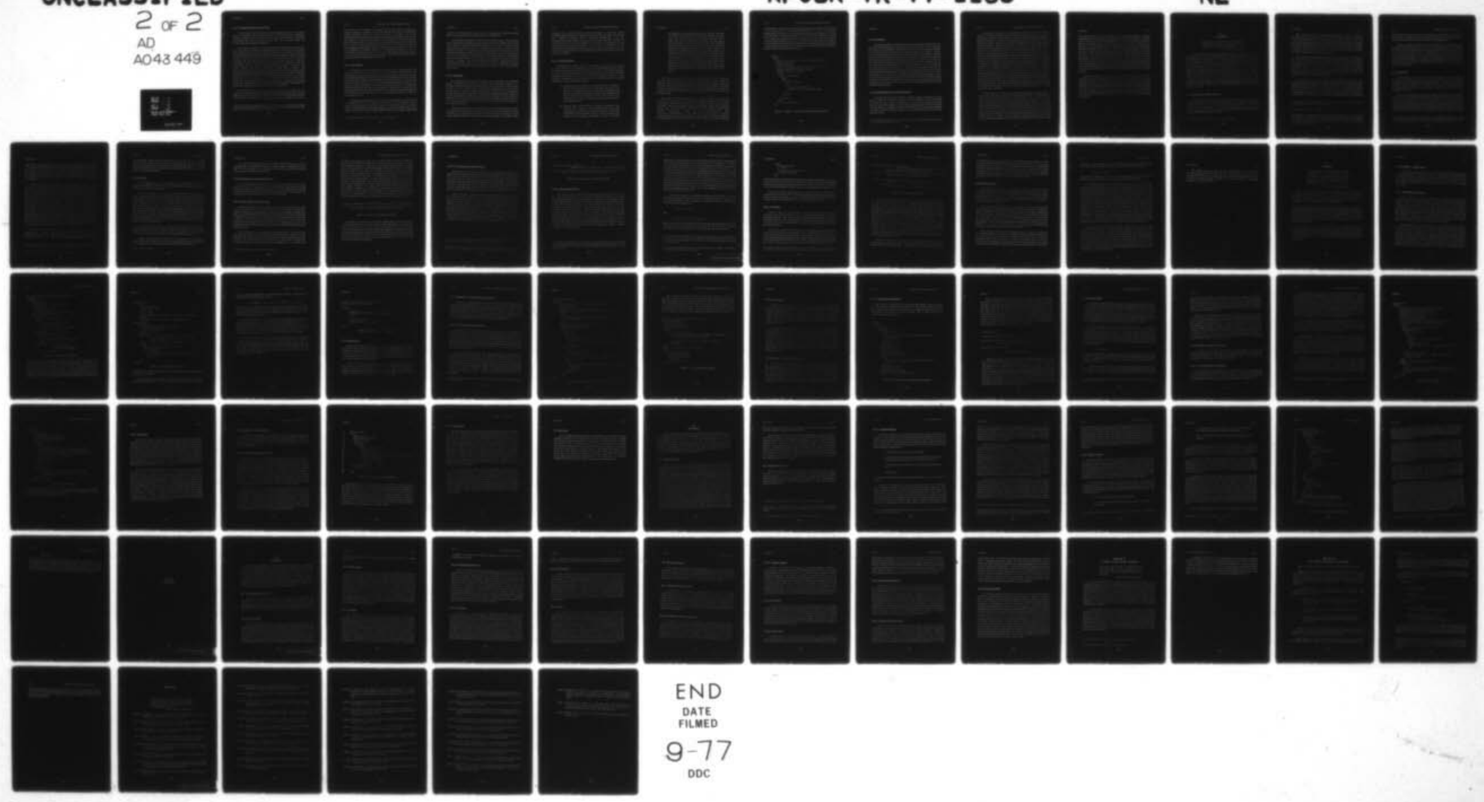
5
Uniformity

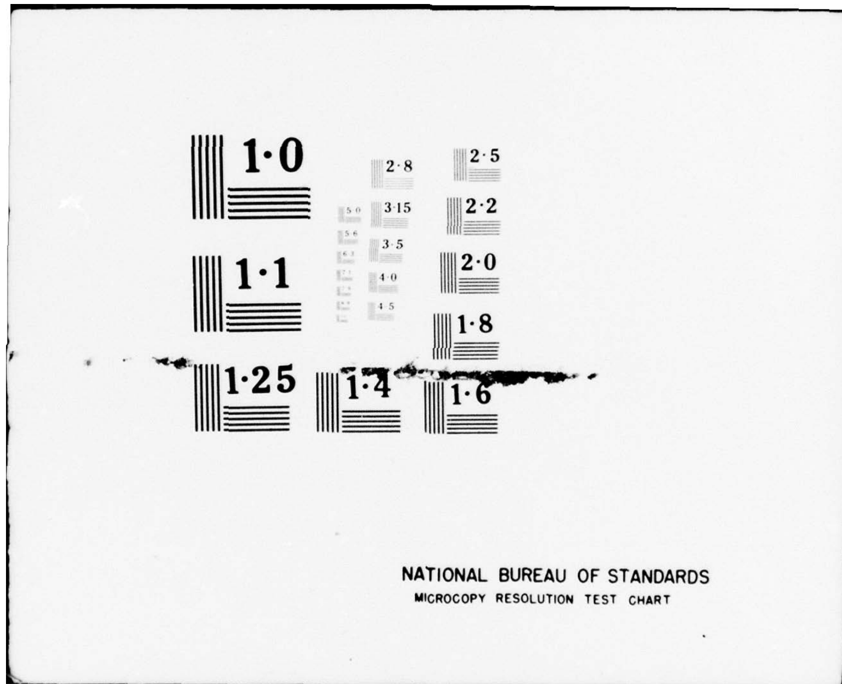
"What's true of one peculiar case is true of all peculiar cases of the same peculiar sort."
- James Thurber, The White Deer

This chapter evaluates the exception mechanism of chapter 4 with respect to the goal of uniformity enunciated in chapter 1. Recall that "uniformity" in this context refers to consistent application of the mechanism within multiple levels of a programming system. We can approach the question in two ways: (1) by demonstrating that the facilities provided by the mechanism work effectively throughout a system, and (2) by showing that the mechanism is largely independent of language-specific semantics. From the former we conclude that the mechanism is indeed useful for exception handling problems that arise at many system levels; from the latter we conclude that the semantics of the mechanism are compatible with languages used at those levels. The utility of the mechanism is best illustrated by examples of 'typical' exception handling situations, whose presentation we defer until chapter 7. In this chaper we concentrate on the relationship of the exception mechanism to other language concepts that may affect the exception handling process.

5.1 *Interactions with the Embedding Language*

We want to show that our mechanism is largely independent of other specific constructs in its embedding language. By doing so we will establish the feasibility of implementing a single set of semantics for exception handling throughout a programming system. (Chapter 7 will demonstrate that those semantics are indeed useful.) We will examine a number of language concepts that interact with exception handling and consider how the mechanism responds to potential variations in their semantics.


AD-A043 449 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
PROGRAM STRUCTURES FOR EXCEPTIONAL CONDITION HANDLING.(U)
JUN 77 R LEVIN F44620-73-C-0074
UNCLASSIFIED AFOSR-TR-77-1136 NL
2 OF 2
AD
A043 449
END
DATE
FILMED
9-77
DDC


1·0
2·8
2·5
3·15
2·2
3·5
2·0
4·0
4·5
1·1
1·8
1·25
1·4
1·6
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

5.1.1 Variable Access and Scope Rules

The permissible forms of access to data strongly influence the expressive power of a language. As a direct consequence, they materially affect verifiability. Naturally, languages used at different system levels and with different verification requirements will specify different access rules. Our mechanism should not interfere with that language prerogative.

The primary interaction of scope rules and the exception mechanism occurs in the handler bodies. A handler requires information from two sources: the signaller that invokes it and the program context with which it is associated. In order to avoid the use of shared global variables, the mechanism permits a signaller to pass parameters to handlers. Of course, if the embedding language permits shared global variables, the exception mechanism in no way hampers their use. However, verifiability may be compromised or complicated in such languages, hence the mechanism supplies a controlled, explicit transmission facility for languages that prohibit implicit, unrestrained communication through global variables. The parameters are not constrained by the mechanism to be value-only, though for some selection policies (e.g. broadcast) var parameters have little utility. Generally, handlers for structural conditions communicate results to the signaller by function invocations (see, e.g., section 7.4.1), and handlers for flow conditions return results through var parameters (see, e.g., section 7.3). Of course, the precise method of communication is determined by the specifications of the module that defines the condition, and it may mix the two techniques freely.[1]

The interaction of the handler and its associated program context is less explicit. The mechanism stipulates only that the handler has (implicit) access to the same data that its associated context has. At first glance, this might seem to require complex manipulations to ensure such access. However, the cost is

[1] Handlers do not have 'return values', largely because var parameters accomplish the same result, allow multiple return values conveniently, require no additional syntax in either signaller or handler, are equivalent in verification difficulty, and allow certain desirable 'defaulting' (see section 7.3.2).

dependent upon the nature of the normal case access control. In BLISS, for example, language restrictions on accessing eliminate the need for an ALGOL-like display mechanism. The entire accessing context of a handler can be (is) represented by a single pointer; loading that pointer into a particular register gives the handler access to the required data in a single instruction. The same is true for BLISS subroutine entry. In full ALGOL-60, however, display management at function entry is more expensive, and handler entry follows suit. Since handlers are treated (and, in general, implemented) much like procedures, we naturally expect the cost of establishing the handler context to be commensurate with the cost of establishing a procedure context. This permits the language design to specify access rules without worrying about complexities that might be introduced by the presence of the exception handling mechanism.

5.1.2 Shared Data

This section and the two that follow it deal with closely related topics. From earlier comments we recall that the exception handling mechanism seeks to accommodate shared data structures. The interesting case occurs when a single structure (instance) is accessible from two or more contexts that are not lexically nested. We have already considered examples of such situations, e.g. the storage allocator example of chapter 4. If a language permits this kind of sharing, then there are natural interactions between the sharing contexts which raise questions about parallelism and its control (synchronization). Thus if the exception handling mechanism interacts with sharing of data, it must interact with parallelism and synchronization as well. We will examine these interactions separately, insofar as it is possible to do so.

A language may not permit sharing of the kind just described, or it may provide a restricted form of it (e.g. Alphard - see section 4.2). The eligibility rules of section 4.6 are intended to cope with whatever form of shared data the language permits. However, if no "interesting" sharing[2] is possible, the eligibility rule simply reduces to a rule that finds a single eligible handler for a particular condition and

[2] "Interesting" means that multiple non-nested references can exist simultaneously.

instance. The mechanism in no way *requires* a language to accommodate shared data, it merely provides sufficiently general rules to deal with it.

The principle underlying the eligibility rules is that of a *user*. The mechanism does constrain the language to supply a sufficiently precise definition of the "user of an instance" that the eligibility determination can be made. In chapter 4 we spoke of the "ability to reference" an instance, and used it as the informal definition of "user". We have also used the term "capability" in a similar way. The essential requirement is that one be able to determine, in a given context, what instances are accessible. The determination may be made partly at compile-time, partly at run-time, with efficiency depending on the degree of run-time checking required. Languages that support information hiding (a requirement for our mechanism) generally support strong typing as well, so the additional restrictions imposed by the exception handling mechanism's definition of 'user' are minimal to non-existent.

5.1.3 Parallelism

At several points in the discussion of the exception mechanism, we noted that certain operations could proceed in parallel. (See especially section 4.7.1). Naturally, some languages do not provide parallel processing, and those that do may provide it in one of several ways. The exception mechanism never *requires* that actions occur in parallel; it merely defines those points at which conceptually independent actions are allowed (if the language and implementation permit) to proceed simultaneously. Wherever previous discussions have used the words "in parallel", purely sequential systems may substitute the words "in undefined order". As we have already noted, certain selection policies may collapse into a single one in a sequential system.

Although the mechanism can take advantage of parallelism if available, it does not define precisely the way in which its actions are to be mapped onto the execution units provided by the language. This has been done deliberately so that several distinct "styles" of parallelism can be accommodated. For example, some

languages may explicitly provide a "process" abstraction. Others may define parallelism implicitly in terms of collateral evaluation. Still other forms are possible (e.g. Hibbard's 'eventual' values in ALGOL-68 [Hibbard 76]). Since the exception mechanism definition states only the possibility, not the necessity, of parallel execution, it does not stipulate how, or even if, the parallelism is to be achieved by the embedding language. In many cases it is sufficient for the programmer to know that he must be prepared for parallel execution; the means by which it is implemented often are irrelevant and need not be related to other language constructs.

5.1.4 Synchronization

Unfortunately, we cannot dismiss synchronization as easily as we did parallel processing. Because we do not want the exception handling mechanism to specify that any particular synchronization facilities be present in the embedding language, we define its synchronization behavior directly (see *sections 4.8 and 4.11*). Although this does not constrain the form of the language facilities, it does introduce complexities in the interaction of those facilities with the exception mechanism. Let us consider two examples.

1) Suppose a function within some module M can raise a condition C under a selection policy that causes the function to wait for the handlers to complete (e.g. broadcast-and-wait). If M expects to operate in a parallel environment, it will very likely provide some mutual exclusion of its functions. Yet if any handler for C attempts to invoke a function of M that requires mutual exclusion, a deadlock may ensue.

2) Suppose that a module M has a function F that may raise structure-class condition C. Suppose further that, under normal circumstances (i.e. when C is not raised), F may be executed in parallel by an arbitrary number of users. However, suppose that, because some consistency test within F fails, C is raised.

> Presumably the same test will fail in all asynchronous executions of F that are manipulating the same (data) instance. This may cause more than one user to reach the raise statement for C. Since the exception mechanism implements its own mutual exclusion requirements, only one user at a time will be permitted to execute the raise. Nevertheless, as soon as one completes it, another may immediately raise the same condition again. This amounts to multiple notifications of the same condition, a complexity that the handlers may not be able (or willing) to cope with. The obvious difficulty here is that the mutual exclusion has occurred too late - the consistency test has already been executed. By enforcing mutual exclusion on the test (with some language construct), we synchronize all executing instances of F (operating on the same data instance), but that unfortunately defeats the original purpose of executing such instances in parallel.

We could devise other scenarios, but the point should be clear. Interactions between synchronization primitives and the exception mechanism are subtle and complex. There are really two questions raised by these examples: does the exception mechanism constrain the form of synchronization primitives, and what language construct(s) do we need in order to avoid the pitfalls illustrated by the examples above? The latter question is considered in section 5.2; we address the former here.

Note that the difficulties that arise in these examples stem from the interaction of the exception mechanism with separately defined mutual exclusion requirements. Nowhere was any mention made of specific primitives used for mutual exclusion; it is the concept that leads to the difficulty. Indeed, we could work through the details of these examples using semaphores [Dijkstra 68], monitors [Hoare 74], critical regions [Brinch Hansen 72], path expressions [Campbell 74], or any other specific technique for achieving mutual exclusion, and we would arrive at the same result. The problem is endemic to the interaction of the two mechanisms. One might leap to the (fortunately

unjustifiable) conclusion that the exception mechanism intolerably constrains the use of mutual exclusion. As a partial rebuttal, figure 5.1 presents an (admittedly unattractive) solution for the second example above, using semaphores as the only synchronization construct. (A solution to the first example appears in section 7.4.1.) We claim therefore that the mechanism does not limit the *forms* of synchronization permitted, but exposes a need for more sophisticated combinations of synchronization and exception handling primitives, combinations that *avoid* these pitfalls. We are thus led to the second question posed in the preceding paragraph, whose answer is the subject of section 5.2.

```
module T
  begin
    condition structural-error policy broadcast-and-wait
    private size:integer
    private sem:mutex
    < other data comprising the fine structure of a 'T' >
    function length(t:T) returns s:integer
      raises structural-error on t
      begin
        if not consistent(t)
          then
            P(t.sem)
            if not consistent(t)
              then
                raise t.structural-error
                < restore 't' to a consistent state >
              fi
            V(t.sem)
          fi
        s ← t.size
      end
    < ... other functions ... >
  end
```

Figure 5.1: Mutual Exclusion under Exceptional Circumstances

5.1.5 Protection

The design presented in chapter 4 has not incorporated any specific features to control the dissemination of information, with two exceptions. First, it respects the principle of encapsulation, and therefore permits no representation information to be transmitted via exceptions unless explicitly sent by the signaller. Second, the mechanism respects the principle of mutual suspicion, and therefore restricts the flow of control during exception handling. These two principles are consistent with the language properties assumed in section 3.1.1. Besides these properties, the mechanism is intended to be neutral with respect to protection problems, but we can offer no demonstration that specific protection problems, e.g. confinement, are not complicated by its presence. It is quite possible that covert channels for leaking information may exist in the exception transmission mechanism, but it should also be noted that the potential recipients of an exception are the possessors of capabilities for an instance. By controlling distribution of those capabilities, a user worried about confinement should be able to limit the spread of information through signalling of exceptional conditions.[3] In general, however, the mechanism does not pretend to address protection issues other than encapsulation and (to a limited extent) mutual suspicion.

5.2 Simplifying the Use of the Mechanism

Although we have claimed that the exception handling mechanism does not impose significant semantic constraints on other constructs in the embedding language, we have observed (particularly in section 5.1.4)) that interactions between constructs may give rise to certain usage paradigms. We may look upon such paradigms as seeking to smooth over difficulties in the use of the exception mechanism and other language elements. Let us briefly consider the reasons such difficulties arise.

[3] If a module has been verified, then its execution behavior is (almost) completely defined by its specifications. This may further bolster its user's confidence.

As illustrated in section 5.1.4, attempts to combine use of synchronization primitives and the exception mechanism can easily produce deadlocks and other undesirable behavior. The example of figure 5.1 suggests that combinations of primitive functions that yield correct behavior may not be immediately obvious or inherently pleasing. Yet it seems reasonable to hope that we can identify certain common situations, such as those cited in section 5.1.4, and "higher-level" operations that directly handle them. After all, such synchronization "primitives" as monitors and conditional critical regions exist precisely because they respond directly to particular usage patterns. Just as we define monitors to provide an access control paradigm connecting synchronization and function invocation, so we can define control flow paradigms connecting synchronization and exception handling. It is beyond the scope of this thesis to explore the space of specific possibilities in detail, but we can consider the example of figure 5.1. The mutex semaphore is intimately connected to the testing of the consistency predicate, which in turn is intimately connected to the raising of the exception. It appears desirable to bundle this diverse collection of language elements, and the control structure that links them, into a single syntactic unit whose semantics would be more immediately understood. Such a unit might be implemented by syntax macros - the method is unimportant - but it should at least be known to the implementation in such a way that its increased readability may be reflected in a more potent or convenient proof rule. We naturally expect that a more intuitive construct should be simpler to verify, and our aim in introducing such paradigms should be to enhance both clarity and verifiability.

Paradoxically, our desire to keep the exception mechanism verifiable has, in some cases, made its operation *less* natural and more difficult to understand. In such situations we wish to introduce constructs that combine language elements because we hope to recapture clarity without sacrificing verifiability. Perhaps the most glaring example is the notion of an abnormal function termination. Assuming that an abnormal completion (e.g. overflow during floating-point addition) is to be reflected through our exception mechanism, we have no machinery at our disposal but to raise some condition. Yet, since raise never alters control flow in the signaller, we must follow the raise with a separate return statement to achieve the desired effect. Obviously, we could define a trivial language construct that combines the raise and return, but the conceptual difficulties at the handler site are

less easily resolved. If we are raising a flow class condition (this is almost invariably the case for abnormal termination), the (only eligible) handler is attached to the context that invoked the signalling function. The coder will often prefer to think of the handler body as performing some local fix-up actions, followed by a transfer of control (perhaps another abnormal termination). However, the semantics say that the handler termination causes control to return to the signaller, which then performs a return, transferring control back to the calling context where any local transfer of control posted by the handler then takes effect. This is unnecessarily cumbersome - instead we introduce an appropriate construct, defined in terms of the primitives just employed, that looks to the programmer just like the desired abnormal function termination. Such a usage paradigm simplifies understanding, yet its formal definition via the existing mechanism retains verifiability. In fact, if such a construct is intimately known by the implementation (as opposed to a user-written syntax macro), efficiencies in object code size and execution speed can be realized.

We may summarize the preceding discussion by observing the need for "humanely-engineered" extensions to the language that includes our exception handling mechanism. This need arises because certain usage paradigms exist that combine separate language elements to form a consistent, more intuitive construct. We have seen examples that combine either synchronization or function return with exception handlering, and we could easily produce others. Only experience and further investigation will determine what language extensions significantly simplify the task of understanding programs that employ the proposed exception handling mechanism.

6
Verifiability

"Humph! These arguments sound very well, but I can't help thinking that, if they were reduced to syllogistic form, they wouldn't hold water."

- W. S. Gilbert, Ruddigore

In the discussion of preceding chapters we have often referred to the properties of the exception handling mechanism as being influenced by the desire for verifiability. In this chapter we consider the problem of verifying programs that use the proposed mechanism. Since we can only present the verification rules in the context of a specific language, we will rely heavily on the semantics and structure of Alphard, our chosen embedding language (see section 3.1.1). The Alphard verification methodology is well-developed, and since it is our intent to extend it in a natural way, we will first establish some conventions and assumptions that, in effect, embody that methodology informally. While less than rigorous, this short-cut permits us to avoid a rehashing of known technique.[1] We then define predicates to characterize the semantics that were described operationally in chapter 4. Using these predicates, we develop proof rules for the signalling and handling constructs. Finally, we assess the value of this approach and the utility of the resulting rules.

6.1 Assumptions and Conventions

The verification methodology of Alphard utilizes Hoare's axiomatic approach [Hoare 69], and we shall construct our proof rules accordingly. This formulation models naturally the specification of functions as assuming some pre-condition and establishing some post-condition. Of course, other verification methods are likely to work as well, e.g. Dijkstra's weakest pre-condition technique [Dijkstra 76], but our presentation will use Hoare's method.

[1] A summary of the Alphard verification methodology appears in appendix B.

We insist that modules be independently verifiable; that is, to prove a particular module we require only its source text (with assertions) and the specifications (predicates) that define the semantics of the modules it uses. Our proof rules will make it possible to verify the body of a signaller without consulting the code of related handlers, and to verify those handlers separately without knowing the implementation of the signaller. The practical application of verification to systems composed of many modules (as those using the exception mechanism are likely to be) depends crucially on this property.

We assume the existing Alphard rules for accessing variables. The details of the rules need not be presented; it suffices to understand their motivation. When we verify a program, we need to be able to identify all distinct variables it manipulates. Both parameter binding (for procedure calls) and ref variables (pointers) make this a non-trivial task, since they introduce the possibility of *aliasing*.[2] Languages that claim to be verifiable (e.g. Euclid [Lampson 77]) have been forced to deal explicitly with the aliasing problem, and the scoping and parameter passing rules of Alphard do just that.

In addition to the existing access rules of Alphard, we impose some restrictions on the syntactic form of handler bodies.[3] We require that the body of a handler consist of a single procedure call. Neither the procedure nor the call has any special syntactic structure; only their existence is prescribed.[4] The actual parameters in the procedure call may be quantities addressible within the handler's associated local context or parameters passed to the handler by the signaller (see section 4.4.2). We do require, as with all procedures, that it be possible to

[2] I.e. two apparently distinct variables (e.g. two formal parameters to a procedure) may actually be the same one (e.g. the same actual parameter was bound to both formals).

[3] We hinted at such restrictions in section 4.5.

[4] We note that this is only a syntactic restriction; any language construct that behaves, for verification purposes, like a procedure is acceptable here. Thus, a syntax macro that possesses the desired properties could be substituted without violating this restriction. We do not require linkage to an out-of-line code body. This requirement has the attractive side-effect of keeping the handler inobstrusive and thereby making it easy for a casual reader of the program to ignore.

determine from the source text which parameters are read and which are written.[5] The notation in subsequent sections presumes this property for all functions and (internal) procedures, as indeed is customary for verification purposes.

Finally, we ignore the verification issues raised by the interaction of the exception mechanism with parallelism and synchronization facilities of the language. Although it might seem that in doing so we are "throwing out the baby with the bath water", we should observe that final disposition of these questions requires a formal definition of parallel processing constructs, which is not yet available for our chosen embedding language. Even with a precise definition, verification of parallel programs remains a difficult, incompletely solved problem and is the subject of considerable current research. A detailed treatment of these issues would exceed the scope of this thesis.

6.2 Predicates

We will define the semantics of our *exception handling constructs* through predicates. In general, these predicates will express the state of the abstraction implemented by a particular module at some instant, e.g. just before or after invocation of some function of the module. The user of the abstraction may presume only the information made available through these predicates when verifying his program.

The existing Alphard methodology already defines the behavior of functions in terms of *pre-* and *post-conditions* (predicates that hold before and after function execution, respectively). We extend the formulation in a natural way to include pre- and post-conditions for exceptions as well. Thus, a module that defines a particular exceptional condition on its abstraction also defines two predicates that hold before and after the condition has been handled. Since we regard handlers as similar to external procedures, it seems natural to specify pre- and post-conditions

[5] This is all that is really needed. We can dispense with the requirement that the handler body be a single procedure call at the expense of having to determine more circuitously the variables it manipulates. However, the procedure call requirement keeps the handlers small, as already noted, and simplifies the proof rule as well.

that define what the handlers are to do. The crucial difference is that the specification occurs in the *calling* module, the signaller, not the module that defines the handler. This is precisely the reverse of the method of function specification, in which the module that defines the function also supplies the predicates. To accommodate this difference in specification, we are forced to alter the proof rule for procedure calls somewhat in adapting it to handlers. The details are presented in section 6.5.

Thus our primary tool in verifying the use of the exception mechanism will be the pre- and post-conditions on each condition name.[6] In subsequent sections we will use these predicates to construct proof rules for both signalling and handling sites, much as we do for function pre- and post-conditions and definition and call sites. In following the function verification paradigm, however, we encounter the phenomenon of using a single predicate for proofs in both the abstract and concrete domains. That is, when we prove that a function satisfies the specifications of its pre- and post-conditions, we are proving its behavior within the concrete domain (i.e. in terms of the representation it manipulates). However, at the call site, we are using the predicates to define the behavior within the abstract domain, since only the abstraction is available to the user of a module. For exception handling, the identical problem arises but, because of the inverted nature of the specifications (discussed in the preceding paragraph), the semantics of the *concrete* domain are required to verify the signalling site, and the semantics of the *abstract* domain apply at the handling site. For both function invocation and exception transmission we need to be able to transform predicates written to describe one domain so that they are suitable for use within the other.

Alphard solves this problem for function invocation by the use of a 'representation function',[7] which establishes the correspondence between elements of the two domains. We appeal to the same mapping for the purposes of exception transmission. For simplicity, we will omit explicit references to the

[6] The unfortunate double meaning of the word "condition" is historical. After section 6.3 predicates will be self-identifying as pre- or post-conditions and the ambiguity of "condition" will be eliminated.

[7] Here, "function" has the traditional mathematical interpretation of a mapping.

representation function when describing the proof rules for the exception mechanism. We will assume that, where necessary, the pre- and post-conditions have been transformed to express properties of the appropriate domain. In light of the preceding discussion, it should always be clear from context which form of a particular predicate is required.

6.3 Notation

In the remaining sections of this chapter, we develop proof rules for the exception mechanism. The notation is conventional; for reference purposes we briefly define the basic symbols used.

The lower case letters x, y, z, a, v are all used to denote sets of variables. x, y, and z always denote the formal parameters of a function that may be changed by the execution of the function,[8] v denotes the formal parameters whose values are inspected but not changed. a denotes a set of actual parameters that may be legally substituted for x, y, or z, i.e. parameters whose values may be changed. Elements of a are thus simple variables, not expressions. e denotes expressions that may be substituted as actual value parameters. Any of these set names may be subscripted. Since the values of a may change across a function call, we will follow the usual convention and write a' to denote the "previous values", i.e. the values before the call. It is occasionally necessary to refer to sets of variables that do not participate in an operation. When the need arises, we will denote a set of these "constant" variables by k.

Function names are denoted by f and g. Thus a formal function specification might be $f(x;v)$, an actual call might be $f(a;e)$. In this notation the semi-colon instead of the usual comma reminds us that the parameters are sets. By assumption, members of a never appear in the expressions of e in such a call.

Condition names are always denoted by c, and handlers are denoted by h. Since handlers, by assumption, are procedure calls, we will often write $h(a;e)$ as a specific handler body. The procedure itself will naturally be written $h(x;v)$.

[8] The so-called "var" parameters.

Pre- and post-conditions are denoted by B^{pre} and B^{post}, respectively. These will generally be subscripted by function or condition name to distinguish them, e.g. B_f^{pre} is f's pre-condition. Other predicates (verification conditions) are denoted by the upper case Roman letters P, Q, and R.

6.4 Proof Rules for the Signalling Site

We can now present the proof rules for the raise statement. For reasons noted in section 6.1, these rules do not express the sequencing of handler invocation except as it relates to the signaller. Thus, synchronization interactions between handlers invoked (potentially) in parallel are ignored in this treatment. We examine each of the selection policies of section 4.7.1 in turn.

6.4.1 The Broadcast-and-Wait Policy

This policy induces behavior extremely close to a procedure call. All eligible handlers are invoked, the signaller waits for all of them to terminate, then execution continues following the raise statement. Indeed, if only one handler is eligible, the effect is precisely that of a procedure call. The proof rule for broadcast-and-wait is slightly more complex than the procedure call rule because it must ensure the "condition persistence" property discussed in section 4.7.1.[9] This is the only variation (for this policy) from the procedure rule. Because the rules for the remaining selection policies are all derived from this one, we will briefly examine its intuitive meaning.

Refer to figure 6.1. In the implication above the line (i.e. in the premise), the first term of the consequent merely requires that the pre-condition on c be satisfied when the raise occurs, a consequence of the condition persistence principle. The second term has two purposes: to complete the statement of condition persistence and to force the post-condition on c to establish the

[9] Indeed, all proof rules for the various selection policies must do so.

necessary state after the raise. Let us consider the implications separately. $B_c^{post}(w,a',e) \supset B_c^{pre}(w,e)$ establishes condition persistence by ensuring that the order of handler execution is irrelevant. Since no manipulation of variables (w) that a handler can perform is permitted to invalidate the pre-condition on c, no handler will be invoked by this raise with a false pre-condition. Viewed somewhat differently, this implication states an invariance about the pre-condition B_c^{pre}. However, note that the invariance is conditioned by the truth of P; it may not (probably will not) hold globally. The other implication, $B_c^{post}(w,a',e) \supset Q(w,a',e,k)$, seems to require that B_c^{post} be strong enough to ensure Q, but this is patently impossible in general, since Q makes statements about variables (k) that do not even appear in B_c^{post}. Once again, we must recall that this implication is conditioned on P, which, by the definition of k, makes the same statements about k that Q does. In this context, the requirement is a reasonable and natural one. Obviously, we must be able to "carry across" the raise statement any information about the signaller's context that cannot possibly be affected by the handlers.

$$\frac{P(a,e,k) \supset [B_c^{pre}(a,e) \wedge \forall w[B_c^{post}(w,a',e) \supset. Q(w,a',e,k) \wedge B_c^{pre}(w,e)]]}{P(a,e,k)\ \{\underline{raise}\ c\ \text{<under broadcast-and-wait>}\}\ Q(a,a',e,k)}$$

Figure 6.1: Proof Rule for 'Broadcast-and-Wait'

Sometimes the procedure call proof rule is written without explicit reference to k, but an axiom of the verification system (called "adaptation" or sometimes the "frame" axiom) permits it to be extended to the form of figure 6.1. We prefer to retain the expanded form because it emphasizes what can be altered during the exception-handling process and what must remain constant. Of course, we could, without loss of generality, eliminate the k's in our proof rules, but we feel that the control of variable accessing during exception handling is of central importance, and we prefer to represent it explicitly.

6.4.2 The Sequential-Conditional Policy

Although from an operational viewpoint the sequential-conditional policy appears quite different from broadcast-and-wait, their proof rules are rather similar. This is not too surprising in light of the observations at the start of section 4.7.1 that the *effect* of raising conditions (as distinguished from the handler selection algorithm) under these two policies generally differs only slightly. Indeed, if the predicate associated with the raise statement under the sequential-conditional policy is identically false, then the effect is precisely that of broadcast-and-wait.[10] Naturally, this collapsing of function is mirrored in the proof rules as well.

Refer to figure 6.2. R is the predicate that appears in the source text of the signalling program. Note that P must ensure that R is false initially. We could relax the semantics to permit R to be true initially, and define the raise statement to have no effect in that case. While straight-forward, this uninteresting case clutters the proof rule and, in practice, never arises. Accordingly, we disallow it. The remainder of the consequent comes directly from the proof rule for broadcast-and-wait, except that the condition persistence requirement is relaxed slightly. Because selection and execution of a handler occurs only if R is false, we naturally require condition persistence only in that case. Indeed, the handler that causes R to become true has probably caused the condition to disappear.[11] Once the condition has been eliminated, it is unreasonable to require B_C^{pre} to hold.

[10] Of course, under one policy the handlers execute in parallel while under the other they execute sequentially. However, this difference is not visible to the signaller and cannot alter the effect it perceives.

[11] It might even seem desirable to have: $R(w,e) \equiv B_C^{pre}(w,e)$, but this is too strong. We might be willing to have B_C^{pre} identically true yet have R describe some more selective termination policy than "no more eligible handlers".

$$\frac{P(a,e,k) \supset [\neg R(a,e) \wedge B_c^{pre}(a,e) \wedge \forall w [B_c^{post}(w,a',e) \supset . \; Q(w,a',e,k) \wedge \neg R(w,e) \supset B_c^{pre}(w,e)]]}{P(a,e,k) \; \{\underline{raise} \; c \; \underline{until} \; R \; \text{<under seq-cond>}\} \; Q(a,a',e,k)}$$

Figure 6.2: Proof Rule for 'Sequential-Conditional'

6.4.3 The Broadcast Policy

The proof rule for the broadcast (without waiting) policy appears in figure 6.3. This rule is quite easy to interpret; it states the condition persistence principle and requires that, no matter what the handlers may do (including nothing), P is strong enough to establish Q. This formulation may seem too restrictive, for it requires that signaller and handlers act *nearly independently. However, these are* precisely the semantics we desire! The broadcast policy is normally used to report to the users of an abstraction a purely informational condition. As an obvious example, consider the completion of an I/O operation. The signaller (frequently hardware) raises the condition but requires no explicit action by the user(s) and proceeds without waiting for any "reply". Applying the rule to practical situations, we find that the sets a and w are normally empty, and that frequently $B_c^{pre} \equiv B_c^{post}$. This usually leads to choosing P=Q, and leaves only the trivial proof that $P \supset B_c^{pre}$. In the absence of interaction between signaller and handler, condition persistence is easily assured.[12]

[12] It is possible to relax the proof rule somewhat, but we then risk serious interactions with parallelism and synchronization mechanisms. Since we have chosen to avoid such problems, we prefer to retain the form given in figure 6.3. Its semantics comfortably accommodate the primary application of the broadcast policy - purely informational signals.

which the condition could be raised. For flow class conditions (the example used to motivate this shorthand), such points correspond only to invocations of functions that can raise the condition, but for structural conditions, literally every primitive operation within the context may be a potential point of interruption.[13] For verification purposes we transform (conceptually) the source text so that *every primitive operation during which a condition may be raised has the relevant handler directly associated with it*. Thus we "copy" the handler to every point at which the condition it processes can be raised.[14] In effect, we apply the eligibility rule of section 4.6 to determine the relevant handler for every condition and primitive operation appearing in the program to be verified. The existence of at least one handler for each condition is assured by the assumption that a "default" module-body level handler always exists, even if the programmer fails to write one explicitly (see end of section 4.5).

The second syntactic transformation eliminates local transfer of control. Assuming that we have already performed the first transformation, all handlers are now associated with primitive operations, i.e. function invocations. Thus we have constructs of the following forms:

$$f(x;v_1)\ [c:\ h(y;v_2)] \qquad (\star)$$

and

$$f(x;v_1)\ [c:\ h(y;v_2) \rightarrow \underline{\text{leave}}\ \text{lab}] \qquad (\star\star)$$

where 'lab' denotes some labelled lexically enclosing context within the current function body. For the purposes of our proof rule, the first form is already acceptable. We replace the statements of the second form by the following:

[13] We are considering the most general case, in which parallel activity is present. In a purely sequential system, only operations that can raise the given condition can be potential points of interruption, except for the purposes of invoking module-body level handlers. These latter handlers, in a sequential system, interrupt nothing and thus constitute a special case.

[14] We ignore the trivial problems introduced by scope rules and redefinition of variables. These are easily accommodated by systematic renaming where necessary.

PRECEDING PAGE BLANK-NOT FILMED

```
begin
    private b:boolean
    b ← false
    f(x;v1) [c: h(y;v2); b←true]
    if b then leave lab
end
```

(Purists may wish to extend the set y with b and place the assignment of 'true' to b inside the body of procedure h.) It should be obvious that this transformation preserves the intended operational behavior described in section 4.9. Indeed, the result of this transformation might be viewed as the definition of the notation in (**).

By means of these transformations, we need only write a proof rule for statements of the form (*).[15] This considerably simplifies the task of describing the formal semantics of handlers, but it raises questions about the practicality of the approach. We defer consideration of such questions until section 6.6.

6.5.2 Proof Rule

Refer to figure 6.4. Ignoring the I_h terms for the moment, we see an instance of the procedure call rule. This is merely the proof rule for the function invocation $f(a_1';e_1)$. We also see the requirement that the handler h fulfill the obligations imposed by the signalling module. These are exactly what we expect to have to prove, but we must additionally account for the interaction of h with the surrounding context.

Since we do not know *a priori* whether h will be invoked, our predicate Q must be written to account for that possibility. We accomplish this by requiring h to supply an invariant, I_h, that holds for the manipulations it performs. Q can then be constructed using the knowledge of that invariant. However, as is evident from the proof rule, the invariant covers only a subset of the variables that h can manipulate (a_3, not a_2). Let us examine the ramifications of this formulation of the invariant.

[15] Assuming, of course, that we have proof rules for the other constructs (e.g. leave) introduced by the transformation. In light of the conventional nature of these constructs, the assumption seems quite reasonable.

$$\frac{\begin{array}{c} P(a,e,k) \supset [I_h(a_3,e_2) \wedge B_f^{pre}(a_1,e_1) \wedge \\ \forall y,z\,[B_f^{post}(y,a_1,e_1) \wedge I_h(z,e_2) \;.\!\supset Q(y,a',e,k)]], \\ I_h(a_3,e_2) \wedge B_c^{pre}\ \{h(a_2;e_2)\}\ B_c^{post} \wedge I_h(a_3,e_2) \end{array}}{P(a,e,k)\ \{f(a_1;e_1)\ [c\colon h(a_2;e_2)]\}\ Q(a,a',e,k)}$$

where: $a = a_1 \cup a_2$ and $e = e_1 \cup e_2$,
and members of 'a' do not appear in members of 'e'.
$a_3 = a_2 - a_1$,
i.e. a_3 = the variables that 'h' but not 'f' can change.

Figure 6.4: Proof Rule for a Handler

The proof rule, in effect, says that if I_h holds before f is invoked and the execution of h preserves I_h, then after f completes, I_h will still hold, and we may assume its truth in proving Q. This rule breaks down (technically, becomes unsound), however, if I_h refers to any variable that f may alter (i.e. a variable in a_1), for then I_h might be true initially but be invalidated by f before c is raised. Similarly, I_h might hold after h has completed, but it might become false through subsequent actions of f. If I_h is restricted to the set a_3, however, f cannot invalidate it (since a_1 and a_3 are disjoint, by definition). We stress that these restrictions on the "scope" of the invariant I_h do not affect the actions h may perform; indeed, a_1 and a_2 will frequently intersect and h will often manipulate variables in the intersection. Rather, the effect is to partition the knowledge available after f completes execution: B_f^{post} contains information about the variables in a_1, I_h contains information about the variables in a_3. Because $a = a_1 \cup a_3$, all necessary information is available to Q, and because a_1 and a_3 are disjoint, no conflicting information can result.

Now let us examine the proof rule in full. Naturally, we require that P be strong enough to ensure I_h initially. Of course, P must imply B_f^{pre} as well in order that the invocation of f be legitimate. The final term in the consequent restates the

preceding discussion; namely, regardless of the manipulations performed by f and h, the truth of both f's post-condition and h's invariant (and P, initially) must be strong enough to ensure the truth of Q. The second part of the proof rule is nothing more than the formal statement that h meets the specifications imposed both by the signaller and its invariant. This verification will, of course, involve another application of the procedure call rule. It should be noted that we have omitted the parameters on B_c^{pre} and B_c^{post} to avoid cluttering the formulae.

6.6 An Assessment

This chapter has sought to demonstrate the feasibility of applying formal verification techniques to the exception handling mechanism proposed in chapter 4. At several points in the presentation, however, we have commented that certain simplifications have been made. One might argue that the value of the result is thereby diminished, or perhaps that the whole problem has been "assumed away". This section considers these issues.

Let us dispose of a minor point first. In the interest of clarity we have ignored the provisions of section 4.4.2 that conditions may have parameters. It should be clear, in retrospect, that introducing parameters clutters the notation slightly but incurs no difficulties. Specifically, B_c^{pre} and B_c^{post} can be extended to include lists of variables changed and inspected, and indeed in section 6.4 we have done so. These lists were suppressed in section 6.5 because they obscure the manipulations of the local context, which were of primary interest in that discussion. However, it is easy to interpret the lists x and v in h(x;v) as including the parameters "passed through" from the handler invocation.

We should note that, although the eligibility rule of section 4.6 is perhaps the most complicated part of the operational definition of the exception mechanism, it does not (need to) appear explicitly in the proof rules of this chapter. This is an advantage rather than a deficiency. The first transformation rule (see section 6.5.1) permits a verification system to "reduce the scope" of handlers to a single operation, and in so doing it depends implicitly upon the eligibility rule. That is, the

semantics of the eligibility rule are embodied in the procedure used by the verification condition generator to produce the set of expressions of the form

$$f(x;v_1)\ [c:\ h(y;v_2)]$$

that are to be verified. Thus, in fact, our proofs do rely upon the eligibility rule implicitly, since the above set of expressions is determined by that rule.

A potentially serious objection that could be levelled at our verification approach is impracticality. In many cases the transformations of section 6.5.1 will distribute the same handler to hundreds or even thousands of control points. We may naturally question the practical utility of such a technique in terms of the present mechanical verification technology. However, we should observe in such cases that the set a_2 of variables altered by the handler is likely to be small. Furthermore, for control points at which a_1 and a_2 are disjoint (a frequent situation for handlers associated with a large scope), no "extra" work is induced by the presence of the handler. It should be possible, therefore, to prove certain theorems (lemmas) once and quickly apply them to the majority of cases. Intuitively, the broader the scope of a handler, the less it is able to manipulate variables with impunity at an arbitrary instant. Thus there are few "hard" theorems to be proved and extensive use of lemmas can eliminate continual "reproving" of the same trivial ones. We claim that if the current verification technology does not have the ability to avoid such mechanical trivialities by discovering and utilizing lemmas, then the impracticality lies in present-day verifiers and not our semantic specifications.

An admitted deficiency of the proof rules is their failure to capture certain synchronization semantics of selection policies. Section 6.1 noted the reason for this omission. The problem of verifying parallel programs is a difficult one and distinct from the exception handling issues of this thesis. We suspect, however, that once the problem is solved, the missing formal semantics for selection policies will be easy to supply, since their informal semantics are quite normal for parallel processing. We depend, therefore, on the success of other, independent research to provide the tools necessary to complete the formal definition of our exception mechanism. In this sense, then, we do not regard the absence of synchronization semantics as a serious shortcoming.

We believe, therefore, that we have demonstrated the possibility of constructing an exceptional condition handling mechanism that is amenable to formal mechanical verification. We consider this mechanism to be an important contribution to language design because it enhances the expressive power of a language without compromising verifiability.

7
Adequacy

"I merely wish to state, avow, affirm, asseverate, maintain, confess, proclaim, protest, announce, vouchsafe, and say that there are precisely ten such tales in all, and each and every one duplicates, substantiates, corroborates, and proves each and every other."

- James Thurber, The White Deer

Having seen the justification of the exceptional condition mechanism on methodological and verification grounds, we now demonstrate its practical applicability. We will show by example that the mechanism accommodates in a natural way several exception handling problems that arise in "real-world" systems and that are not successfully handled by existing mechanisms.

In section 3.3, three exception handling problems were presented as examples of "real-world" situations that the mechanism should be able to solve. In this section we examine each of these problems in turn, as well as some others that illustrate specific features of the mechanism. The presentation for each example includes a prose description of the problem, a coded solution using the exception handling mechanism, and a commentary that discusses some of the more subtle aspects of the solution.

The problems as posed here are, in some cases, slight simplifications. Because many interesting exceptional conditions have far-reaching effects within a system, we have been obliged to pare down the examples in order to emphasize the central issues they address. In no case, however, has an example (knowingly) been restricted because the exception mechanism is incapable of handling a more complex case.

7.1 Example 1: Symbol Table

For our first example we present a straight-forward symbol table module, with a few minor twists. The symbol table is essentially a content-addressed association structure of bounded size, with functions that add a <name,value> pair and retrieve a previously entered pair. We consider this example because it is self-contained and illustrates some basic aspects of the exception handling mechanism.

7.1.1 The Symbol Table Problem

Refer to figure 7.1. The specifications for a *form* 'symbol-table' appear in a style close to that used for Alphard. In fact, our symbol table example is an altered version of the one in [London 76], with the block-structured aspects removed and the exceptional conditions added. Briefly, the specifications define a 'symbol-table' to be a set of pairs <s,v>, where the first elements of the of the pairs are all distinct. Further, the cardinality of the set is limited to at most 'n'. Note that both condition and function specifications are included, with pre- and post-conditions expressed in the abstract domain. When pre-conditions are omitted, they are assumed to be identically 'true', when post-conditions are omitted, they are assumed to state that all parameters remain unchanged.

By interpreting the pre- and post-conditions in operational terms, we discover that the 'lookup' function leaves its arguments untouched, but may raise either 'absent' or 'present'. In the former case no pair with the desired first element exists in the table. In the latter case, however, such a pair does exist, and the parameter to 'present' is the pair's second element. Thus 'lookup' *always* raises a condition. Contrasting this specification with the more usual form of 'lookup' (as a value-returning function), we see that what would otherwise appear as a part of lookup's post-condition shows up instead as 'present's pre-condition. This is intuitive and comforting. The situation for 'insert', however, is somewhat different.

```
form symbol-table(n:integer,T:form<=,←>,V:form<=,←>)=
  beginform
  specifications
       requires n ≥ 1;
       let symbol-table = assoc:{<s:T,v:V>};
       invariant
         cardinality(assoc) ≤ n
         ∧ t1,t2∈assoc ⊃ (t1.s=t2.s ⊃ t1.v=t2.v);
       initially symbol-table = { };
       functions
         lookup(st:symbol-table,str:T)
           raises
             absent on lookup policy broadcast
               pre = t∈st ⊃ t.s≠str,
             present(v:V) on lookup policy broadcast
               pre = ∃t∈st st t.s=str ∧ t.v=x
           endraises,
         insert(st:symbol-table,str:T,val:V)
           raises
             full on insert policy broadcast
               pre = cardinality(st)=n ∧ (t∈st⊃t.s≠str)
           endraises
           post normal = if ∃t∈st st t.s=str
                             then st=st'-{t}U{<str,val>}
                             else st=st'U{<str,val>}
           post full = st=st';
```

Figure 7.1: Symbol Table Specifications

Here the normal pre-condition, which would prevent 'insert' from being invoked when the symbol-table is full, is relaxed to permit unrestricted calls. The former pre-condition for 'insert' now becomes the pre-condition for 'full' instead, suggesting that a state that was previously assured by a verification condition will now be assured by an explicit test within the function. We recognize this "delaying the binding" as a means of ensuring *operationally* correct behavior without sacrificing formal correctness. Introducing exceptional conditions that reflect an invalid state

```
representation
      unique
        names:vector(T,1,n),
        values:vector(V,1,n),
        last:integer
          init last←0;
      rep(last,names,values) = {<names[i],values[i]> | i∈[1,last]};
      invariant
        last∈[0,n] ∧ ((i,j∈[1,last]∧i≠j) ⊃ names[i]≠names[j])
implementation
      body lookup =
        first j:upto(1,last) suchthat names[j]=str
          then raise present(values[j])
          else raise absent;
      body insert
        out normal = ∃i∈[1,last] st (names[i]=str ∧ values[i]=val ∧
                  ∀j∈[1,last'] ∃k∈[1,last] st
                     (names[j]'=names[k]∧i≠j ⊃ values[j]'=values[k]))
        out full = last'=n ∧ ∀i∈[1,last'] names[i]≠str
        begin
          first j:upto(1,last) suchthat names[j]=str
            then values[j]←val
            else if last<n
                then last←last+1; names[last]←str; values[last]←val
                else raise full
               fi
        end
endform
```

Figure 7.2: Symbol Table Implementation

at function invocation normally shifts (part of) the function's pre-condition to the exception's pre-condition.

We note that two post-conditions appear for function 'insert'. This notation merely separates the part of the post-condition that applies in the "normal" case

from the parts that apply when exceptions have been raised. The actual post-condition is a disjunction of these terms, i.e.

$$(\neg\ \text{raised(full)} \wedge \text{post normal}) \vee (\text{raised(full)} \wedge \text{post full})$$

where 'raised' is a predicate that is true if and only if its argument condition was raised by the function 'insert'. By separating the terms this way, we see more easily what actions occur under various exceptional cases without having to wade through long, disjunctive predicates.

Now refer to figure 7.2. The remainder of the Alphard form contains the representation information and implementation of the symbol table functions. The only interesting aspects of these parts are the use of the raise statement in the manner described by the predicates in figure 7.1 and the divided output assertion for the body of function 'insert'. The two portions of this assertion correspond in the obvious way to the parts of the post-condition in the abstract specification.

The code fragments of figure 7.3 illustrate possible uses of 'lookup' and 'insert'. They adhere only loosely to the Alphard syntax; specifically, the separator ':' within square brackets has none of the usual binding semantics and is used only to separate the component parts of a handler. The invocation of 'lookup' sets 'v' to the value associated with 's', if any, or zero if 's' does not appear in 't'. The invocation of 'insert' expects to enter the pair <r,w> into 't', but will leave the block labelled 'l' if the table is full.

```
shared t:symbol-table(47,string,integer)
unique r,s:string, v,w:integer
. . .
l:begin
        < set 's' >
        lookup(t,s) [present(x): v←x | absent: v←0]
        < use 'v' >
        . . .
        < set 'r' and 'w' >
        insert(t,r,w) [full:    → leave l]
        . . .
  end
```

Figure 7.3: Sample Use of Symbol Table Form

7.1.2 Assessment

This example demonstrates the most elementary use of exceptional conditions - to specify unusual returns from a function. Although we have altered the obvious specification slightly through the introduction of the condition 'present', the solution remains, in essence, a finger-exercise. Yet it illustrates a point raised in section 1.1, namely, that what constitutes an exception can (should) be determined by the user of an abstraction, not its implementor.

In most of the remaining examples, we will dispense with a full-blown Alphard form, which becomes unnecessarily verbose for our purposes, and adopt a more conventional syntax for some of the language constructs. We do so to highlight the exception handling aspects of the examples, and to play down possibly unfamiliar aspects of Alphard that are not essential to exception handling.

7.2 Example 2: Inconsistent Data Structures

Section 3.3.1 suggested that the exception mechanism should handle errors arising from memory failure. We can generalize the notion to include all situations in which an inconsistency arises in the internal data structure of a module and the condition cannot be repaired by inherent redundancy. Certainly a parity error fits this classification[1], since the bit pattern in a memory word is demonstrably invalid, but the correct intended pattern is unknown by the memory system. We will build upon the symbol table example above to illustrate this class of errors.

7.2.1 The Inconsistent String Problem

We suppose that, as in the previous example, we have a symbol table that associates strings and integers. Suppose, too, that whenever the content of a string is retrieved, the 'string' module checks its representation for validity. If the string is invalid (e.g. its length field is negative), the string module raises a condition 'bad-string', which is naturally fielded by the symbol table module. We will alter the previous version of this module to maintain a duplicate vector, 'dnames', of the names in the table. We will then add handlers to perform the necessary recovery. Refer to figure 7.4.

The string module defines a function that copies strings ("←") and a function that tests two strings for equality. Other functions of the module, which would manipulate the actual contents of strings, are not shown. If the comparison function "=" detects any inconsistency, it raises 'bad-string' on the inconsistent (instance of) string. Since 'bad-string' is structural, it goes to the user of the particular string (in this case, the symbol table module). If, upon return from the handler, the string is now valid, "=" continues; otherwise, the string is reinitialized and the condition 'reset-string' is raised. '=' performs identical actions for each of its arguments.

[1] In the absence of error-correcting codes, of course. Assume that whatever codes exist have already been applied unsuccessfully.

```
module string(n:integer)
  begin
        private length:integer, chars:vector(character,1,n)
        init length←0
        condition reset-string policy broadcast
        condition bad-string(var s:string) policy broadcast-and-wait
        function ←(s1,s2:string)
          begin
          for i:upto(1,s2.length) do s1.chars[i]←s2.chars[i] od
          s1.length←s2.length
          end
        function =(s1,s2:string) returns b:boolean
          raises reset-string on s1,s2
          raises bad-string on s1,s2
          begin
            if s1.length<0
              then
                raise s1.bad-string(s1)
                if s1.length<0
                  then s1.length←0; raise s1.reset-string; return fi
              fi
            if s2.length<0
              then
                raise s2.bad-string(s2)
                if s2.length<0
                  then s2.length←0; raise s2.reset-string; return fi
              fi
            if s1.length=s2.length
              then
                first i:upto(1,s1.length) suchthat s1.chars[i]≠s2.chars[i]
                  then b←false
                  else b←true
              else b←false
              fi
          end
  end
```

Figure 7.4: Form 'String' with Exception Detection

Now consider the altered version of 'lookup' that appears in figure 7.5. (We have used a less terse syntax to show the proper placement of handlers.) Note especially the 'fixtable' routine, which removes the clobbered entry from the table and notifies the users of the symbol table that an entry has been lost. To accommodate this additional condition, the post-condition of 'lookup' would have to be extended. We omit the details, observing only that the invariants as they appear in figures 7.1 and 7.2 are still valid despite the manipulations of 'fixtable'.

```
condition lost-entry policy broadcast-and-wait
function lookup(st,s)
  raises lost-entry on st
  raises absent on lookup
  raises present(v) on lookup
  begin
       for i:upto(1,last) do
         if =(names[i],s) [names[i].bad-string(str): str←dnames[i] ]
           then raise present(values[i]); return
           fi
         od [names[i].reset-string: fixtable(st,i) → return ]
       raise absent
  end
routine fixtable(st:symbol-table,i:integer)
  begin
       names[i]←names[last]
       dnames[i]←dnames[last]
       values[i]←values[last]
       last←last-1
       raise st.lost-entry
  end
```

Figure 7.5: Re-implementation of 'lookup'

7.2.2 Assessment

In this example we see a number of features of the exception mechanism not present in the 'pure' symbol table example. Most significant is the communication between signaller and handler for 'bad-string', which illustrates a common style of recovery. Note that the recovery actions apply to the 'names' data structure only; if the string module raises a condition on string 's', no handlers in 'lookup' are eligible. The caller of 'lookup' is a user of 's', since he supplied it as a parameter. 'lookup' relies on higher-level functions to take appropriate recovery actions if 's' becomes inconsistent.

The observant reader will have noticed that 'lookup' assumes that '←' (in the string module) raises no conditions. In practice, of course, it might, since this implementation of strings imposes an upper bound on their length. We have deemed it prudent to avoid this additional complexity, since the implementation is less than ideal anyway. Frequently, when a careful examination of a module reveals a large number of exceptional conditions, a different implementation (with a more robust structure) will cut down the potential sources of trouble. Such an implementation will produce a module that is considerably easier to use. Specifically, this module would better serve its users by an implementation that allowed assignment of arbitrary length strings.

7.3 Example 3: *Arithmetic Exceptions*

The preceding example illustrates an important aspect of exception handling: the transmission of recovery information between signaller and handler. Unfortunately, the communication protocol is somewhat obscured by the details of the recovery actions. In this example we strip away some of the complications and side effects that are frequently associated with recovery and concentrate on the communication details. We consider arithmetic exceptions because they admit a variety of reasonable recovery actions while requiring the maintenance of minimal internal state. The interaction of signaller and handler is thus directly emphasized.

7.3.1 Floating-Point Underflow

We omit the implementation of the floating-point arithmetic form, presenting only its (incomplete) specifications in figure 7.6. The complete specification for this module would naturally include additional exceptions and functions, but the specifications given will suffice for this example. Let us briefly interpret the specifications in operational terms.

```
form fp
  beginform
  specifications
        functions
          add(a,b:fp) returns c:fp
            raises
              overflow(var v:fp) on add policy broadcast-and-wait
                pre = a+b>maxfp
            endraises
            post normal = c=a+b
            post overflow = c=v,
          maxfp returns c:fp
            post = ∀a a:fp ⊃ c≥a,
          minposfp returns c:fp
            post = ∀a a:fp∧a>0 ⊃ 0<c≤a,
          div(a,b:fp) returns c:fp
            raises
              underflow(var v:fp) on div policy broadcast-and-wait
                pre = 0<|a/b|<minposfp
            endraises
            post normal = c=a/b
            post underflow = c=v,
          sign(a:fp) returns c:integer
            post = if a=0 then c=0 else c=a/|a|
          . . .
```

Figure 7.6: (Partial) Floating-Point Module Specifications

Two constant-valued functions, 'maxfp' and 'minposfp', define respectively the largest and smallest positive values that the 'fp' form can manipulate. These semantics are clearly given by the post-conditions on the functions. We use these functions in both the specifications of other functions of the 'fp' form (e.g. 'add' and 'div') and in the modules that use the 'fp' abstraction (see figure 7.7). In the 'add' function's specifications, 'maxfp' appears in the definition of the 'overflow' exception, whose pre-condition states that 'overflow' is raised if the (real) sum of the arguments to 'add' exceeds the implemented range (defined by 'maxfp'). Similarly, the pre-condition on 'underflow' (for the 'div' function) states that the exception is raised if the quotient is not identically zero but is too small to be represented, i.e. is less than 'minposfp' in absolute value.

```
private x,y,z:fp

z←add(x,y) [overflow(v): v←maxfp]

z←div(x,y) [underflow(v): if sign(x)=sign(y)
                    then v←minposfp else v←-minposfp fi ]

z←div(x,y) [underflow(v): v←0]
```

Figure 7.7: Use of Form 'fp'

Now let us examine the interaction of the exceptions with the functions that raise them. Both 'add' and 'div' have the same structure in this regard; we consider 'add' *specifically*. By definition, 'add' always returns a value 'c', but the computation of 'c' depends (possibly) on the raising of the 'overflow' condition. If 'overflow' does not occur, then the "normal case" post-condition applies, stating that 'c' is merely the desired sum of the parameters to 'add'. However, if 'overflow' <u>does</u> occur, the post-condition tells us that 'c' is set to whatever value 'v' has. We see from the definition of 'overflow' that 'v' is a (<u>var</u>) parameter passed to the handler of the 'overflow' condition. Thus, in the event of overflow, the handler has the opportunity (duty) to determine the value of the operation 'add'. For some examples of possible handler actions, refer to figure 7.7.

7.3.2 Assessment

We must not forget that the operations performed by the handler in order to compute the desired return value occur in the abstract domain. It is partly for this reason that the functions 'maxfp' and 'minposfp' exist, for the handler requires a representation-independent way to obtain the largest and smallest positive floating-point numbers. Even the *zero* appearing in the second underflow handler is in fact an abstract version of the floating-point zero. In Alphard quantities that are constant-valued functions may be defined by some abbreviated notation, but it is the concept that they are constants with *abstract* properties that matters.

We note that the actions taken by the handlers in figure 7.7 are rather conventional. It might be more convenient if the specifications of form 'fp' indicated a frequently desirable 'default' behavior, so that handlers might be omitted if the 'default' actions is acceptable to the user. For example, to make the default action on underflow be 'result set to *zero*', we need only add a term to the pre-condition stating 'v=0'.[2] Now the handler is relieved of the duty of setting 'v' to *zero* (in the third example of figure 7.7). The handler body of that example is then null and the enabling construct

```
[underflow(v):   ]
```

may be omitted entirely. Obviously, appropriate default values may be introduced for other conditions as well. While this is not a general defaulting mechanism, it can improve readability of the source text by cutting down on the number of handlers that must be included.

Handlers may wish to employ more elaborate recovery strategies than those illustrated in this example. Underflow in a particular floating-point division might necessitate recomputation using arithmetic that offers, say, a wider range of

[2] Naturally, in the implementation part of form fp, we must also insert in the function 'div' the assignment 'v←0' just before the 'raise underflow(v)'.

representable real numbers. To perform this more accurate arithmetic, another form would be used and it would have to supply transfer functions to convert between regular floating-point numbers and its more complicated representation. We have omitted examples of such usage precisely because the additional state manipulation blurs the essential features of the communication between form 'fp' and the handlers of its 'underflow' condition.

We note in passing that the concept 'overflow' may be applicable to more than one function, e.g. 'add' and 'div'. Flow-class conditions with identical names may be specified for different functions of the same module without wreaking havoc with verification at the handler site. The syntactic transformations of section 6.5.1 will eliminate any possible ambiguity, though they may as a result associate identical handler text with different operations. As long as the handler's pre-condition is weak enough that it is implied by the exception's pre-condition, no inconsistency arises. While such a "trick" might seem to compromise clarity, in practice it can reduce the number of separate conditions defined by a module, thereby producing a more concise specification.

7.4 Example 4: Resource Allocation

In section 3.3.2 we briefly mentioned some of the possible exceptions that might be defined by a resource allocator. We also toyed with such exceptions in section 4.6, promising to return with a fuller treatment. Let us now consider the exception handling behavior of an allocator in detail.

7.4.1 The Storage Allocation Problem

For definiteness, we suppose we are specifying a primary memory allocator. The allocator creates and deletes storage segments specified by descriptors. We ignore the details of descriptors, noting only that they contain a size field that can be manipulated by the allocator (by calling the 'descriptor' module, of course).

The allocator provides an abstraction called a 'pool', whose maximum capacity is established at instantiation time. (This is not a crucial property, but it simplifies the exposition.) It also provides two functions, 'allocate' and 'release'. 'Allocate' accepts a pool instance and an integer size as parameters and creates a descriptor for a segment of the desired size, deducting resources from the specified pool. If adequate resources are not available to create the segment, the condition 'pool-low' is raised. After the handlers for this condition have freed what resources they can, allocation is again attempted and, if unsuccessful, the condition 'pool-empty' is raised and the allocation request fails. The 'release' function has no exceptional behavior - it merely returns resources from a segment to the pool.[3]

We assume that the allocator is to execute in a fully parallel environment and must therefore provide its own internal synchronization. We use mutual exclusion semaphores for this purpose. Refer to figure 7.8 for the code of the allocation module. We offer some critical comments below; first, however, let us consider figure 7.9, which contains a hypothetical user of the allocator.

The module 'arb' is intended to illustrate a possible use of the allocator and performs no intuitively interesting operations. A caller of 'f' supplies units of information to be stored in a data base of some sort. The function 'f' copies these units into a segment, which it obtains from the allocator, then creates a second segment into which associated retrieval information (computed by 'f') is stored. Both segments are then entered into a complex association structure, namely 'm', by the unspecified procedure 'add-to'. The 'arb' form would normally contain other functions besides 'f' - we have omitted them for clarity.

'Arb' claims to be prepared to handle 'pool-low' at any instant. Indeed, except during the time when segments are being added to 'm', 'pool-low' is handled by the 'squeeze' procedure, which laboriously compacts the structure 'm' and releases unnecessary elements to pool 'p'. However, it is crucial that the two segments be entered into 'm' indivisibly with respect to compaction, hence the invocation of 'squeeze' is inhibited (by a masking handler for 'pool-low' that does

[3] We have omitted the treatment of a potential third condition 'pool-below-threshold' (see section 3.3.2) to avoid cluttering the example. Two conditions adequately illustrate the desired interactions.

```
module pool(n)
  begin
        condition pool-low policy sequential-conditional
        condition pool-empty policy broadcast
        private outer,inner:mutex, free:integer
        < additional fine structure of a 'pool' >
        function initialize(p:pool) =
            (p.free←n; <initialize fine structure>)
        function allocate(p:pool,amt:integer) returns d:descriptor
          raises pool-low on p, pool-empty on allocate
          begin
            macro adequate(p,amt) = p.free≥amt $
            P(outer); P(inner);
            if not adequate(p,amt)
              then
                V(inner)
                raise pool-low until adequate(p,amt)
                P(inner)
                if not adequate(p,amt)
                  then V(inner); V(outer); raise pool-empty; return
                  fi
              fi
            p.free ← p.free - amt
            <  create descriptor 'd' for segment of size 'amt'  >
            V(inner); V(outer)
          end
        function release(p:pool,d:descriptor)
          begin
            P(inner)
            p.free ← p.free + <size of segment referenced by 'd'>
            <release space associated with 'd'; reset size of 'd' to zero>
            V(inner)
          end
  end
```

Figure 7.8: Storage Allocator

```
module arb
  begin
        shared p:pool
        private m:hairy-list-structure
        condition nosoap policy broadcast
        function f(ct:integer,<info>)
          raises nosoap on f
          begin
            private d1,d2:descriptor
            macro local-cleanup = release(d1); raise nosoap $
            d1←allocate(p,ct) [pool-empty: raise nosoap → return ]
            d2←allocate(p,ct+47) [pool-empty: local-cleanup → return ]
            <  fill in d1 and d2 using <info>  >
            begin
              add-to(m,d1); add-to(m,d2)
            end [pool-low:   ]
          end
        routine squeeze
          begin
            <  perform data-dependent compaction of 'm', using
               'release(p,d)' to release any elements 'd' removed
               by compacting 'm'  >
          end
  end  [pool-low: squeeze()]
```

Figure 7.9: Use of Form 'pool'

nothing) while they are being entered. The handlers for 'pool-empty' are straightforward - they merely translate the condition name to one appropriate to the 'arb' abstraction and propagate it to the caller of 'f'.

7.4.2 Assessment

The code in figures 7.8 and 7.9 illustrates the interaction between the exception mechanism and semaphores for synchronization. Were it not for the existence of condition 'pool-low', a single semaphore would suffice to ensure mutual exclusion of invocations of 'allocate' and 'release' on the same pool. Because we must permit 'release' to be invoked as a result of raising 'pool-low', two semaphores are necessary, one to do the normal mutual exclusion ('inner'), the other to prevent re-entry to 'allocate' ('outer'). The naive approach, with a single semaphore protecting the bodies of the two functions, leads to a deadlock when 'pool-low' is raised and 'release' is invoked from a handler. It would be desirable if more sophisticated constructs were available to permit a more natural expression of this interaction. (Recall the observations of section 5.2.)

We should note that, although the 'pool' module is prepared to handle parallel activity, 'arb' is not. Multiple instances of 'arb' having distinct 'hairy-list-structure's would execute correctly, but multiple simultaneous instances of 'f' *applied to the same instance of 'arb'* would not function properly. The eligibility rule in such a case would still permit 'squeeze' to be triggered in one instance of 'f' while another was *attempting to perform the supposedly* indivisible compaction of 'm'. To eliminate this difficulty, an explicit 'updating' flag and appropriate mutual exclusion would have to be introduced. In general, replacing the implicit state encoding (i.e. the masking of 'squeeze') represented in an instantaneous flow path by an explicit encoding (i.e. a flag) represented by an element of the data structure leads to a clearer, if slightly more verbose, program. We have simply ignored this (rather commonplace) difficulty by assuming the situations that require such data structures do not arise in our example.

7.5 Example 5: I/O Completion

As we discussed in section 3.3.3, input/output completion may be regarded as an exceptional condition. Although this example overlaps somewhat into the areas of inter-process communication and synchronization, it illustrates that the exception mechanism may apply naturally to situations in which an expected event occurs relatively infrequently.

7.5.1 The Real-Time Update Problem

We recall the example of section 3.3.3. Suppose we have a collection of asynchronous processes, one of which accumulates data from external sensors in an airplane. The remaining processes perform calculations that are priority-ordered, so that if new sensor data arrives, they are to begin anew with the more recent input rather than complete their calculations on the old data. In this way, the most important quantities are calculated as soon as possible when updated information is received. To complicate matters slightly, however, some calculations, once begun, must run to completion. In a practical application this can arise if we need to ensure that a set of values <x,y,z> representing the computed position of the aircraft is calculated using a single buffer of data. Permitting the calculation of this position to be interrupted could result in inconsistent co-ordinate values.

Refer to figure 7.10. The function 'grind1' represents one of the computational processes, which all share an instance 'buf' of the abstraction 'sensor-buffer'. This abstraction is implemented elsewhere; all that concerns us in this example is the structural condition 'new-data', which is raised when new values have been placed in the sensor buffer. The condition is broadcast asynchronously to all users of 'buf' (i.e. the selection policy is broadcast). As results are computed, they are placed in the output structure 'display', another externally-defined abstraction which all the 'grind' functions share. Those results are transmitted by the 'screen' module to a physical output medium.

```
1      module crunchers
2        begin
3          shared buf:sensor-buffer
4          shared display:screen
5          function grind1 =
6            while true do
7              l:begin
8                  private restart:boolean
9                  restart←false
10                 <  perform computation 1  >
11                 <  perform computation 2  > [new-data: restart←true]
12                 if restart then leave l fi
13                 <  perform computation 3  >
14                 . . .
15               end [new-data:     → leave l]
16         function grind2 =
17             < similar to the above >
18         . . .
19       end
```

Figure 7.10: Real-Time Computation Module

We identify two styles of computation occurring within the 'grind' functions. The first, typified by line 10, can be aborted anywhere. Note that if 'new-data' is raised when control is at line 10 (or 13), the handler on line 15 will be invoked and will cause control to transfer to the start of the loop. The second style of computation, typified by lines 11 and 12, cannot tolerate abortion. Accordingly, a handler is supplied on line 11 (masking the one on line 15) that prevents transfer of control. However, it sets a boolean variable, which is tested after the computation is finished. In this way, multiple related output values may be computed without fear of interruption and subsequent inconsistent display.

7.5.2 Assessment

We have ignored the fine structure of the computations because it is largely irrelevant to the exception handling aspects of the example. However, since we are using the mechanism for inter-process communication, synchronization is a relevant issue and the fine structure necessarily involves some synchronization. Specifically, each computation will require accesses to 'buf' to acquire raw input data. These accesses will have to be synchronized within the 'sensor-buffer' module to prevent the asynchronous arrival of sensor input from producing inconsistent values. Assuming this difficulty to be resolved, the 'grind' computations may assume they receive proper data from the buffer. Computations then ensue using the raw input. If all related results are computed before any are output, the 'new-data' condition need be masked (as on line 11) only when those (locally stored) results are actually being transmitted to the 'display'. Thus the eligible life-time of the handler may in practice be very short - the duration of the 'output' phase of a computation.

The explicit masking of an asynchronous condition and use of a boolean to recall its occurrence reminds us of conventional interrupt processing when an interrupt must be held pending. Usually this is done by dedicating a piece of hardware to a polling task, polling for the unmasking of the interrupt. Here we are doing the same thing in software with an explicit test. All software masking is performed this way, by checking at "unmasking time" to see if the condition occurred during the masked interval. The frequency with which this programming construct appears suggests that it might constitute a usage paradigm in the sense of section 5.2. It would be desirable to provide appropriate syntax to handle this common situation.

7.6 Summary

The preceding examples demonstrate the flexibility of the proposed exception handling mechanism as a practical tool in solving "real-world" problems. Although the situations presented have been, in most cases, simplified versions of actual problems that arise in operating systems and application programs, the individual "assessment" subsections have indicated what simplifications were made and how, in practice, they might be overcome. We have not attempted to flesh out the details of every case, but rather to illustrate the power and scope of the mechanism by representative examples. We thereby support the claim that the uniform application of the particular semantics we have chosen for our exception handling mechanism has practical value throughout a programming system.

8
Practicality

As the final stage in our justification of the proposed mechanism of chapter 4, we address the problem of implementation. A detailed description of an implementation would require us to delve into the workings of a specific programming language and would carry us rather far from the primary topic. Besides, the obvious candidate language, Alphard, does not have an implementation to date. To substantiate our claim of practicality, however, we must show informally at least that a reasonable implementation is possible. In this chapter we sketch, in general terms, the main features of an implementation.

8.1 Handler Bodies

Most of the elements of the *mechanism* have obvious implementations. Handler bodies are compiled much the way procedures are (or, if one adheres to the syntactic restrictions of chapter 6, the way procedure calls are). The only significant problem is access to the local variables of the interrupted (associated) context. In an ALGOL-like language with a run-time display, this problem is easily solved - the handler body behaves much like an inner block and the display cells other than the top one are identical to those of the associated context. The top display cell points to the local context of the handler, if *any*. Some slightly non-standard display manipulations may be required at handler invocation, but these are minor details. If the language does not use a display (e.g. BLISS) but *can determine* the location of all variables relative to the top of the run-time stack, the accessing of local context information from within the handler becomes more difficult. Now two disjoint stack regions *must* be accessed, in effect a two-level display. However, since the pointer to the associated context will be available at handler initialization (see section 8.2.1), it is straight-forward to compile handler *code* that accesses the associated context relative to that pointer and its own local variables relative to the top-of-stack pointer. *In fact*, if the handler body is constrained to be a *procedure* call, the former pointer will be necessary only to access the variables

needed as parameters to the call.[1] This implies that the called procedure can be compiled normally - a considerable simplification.

Local transfer of control is also quite straight-forward. We want to achieve the effect of a goto to an appropriate local address upon return from the function invocation that has been interrupted by the handler. If function invocations are implemented as subroutine calls, this amounts to altering the stacked return address. Thus the implementation need only assure that the location of that return address is obtainable when the handler is executing, a requirement that is easily satisfied.[2] We observe that the compiler can determine which function invocations can possibly be interrupted by a handler specifying a local transfer of control, and any additional code required to make the return address available need be generated only for those invocations.[3]

8.2 *Eligible Handlers Set*

The only significant implementation difficulty is the eligible handlers set. Here the goal is to minimize the cost of maintaining the set, perhaps with some higher cost at search time. There are really two parts to the problem: maintaining the 'enabled handlers set' (in the terminology of section 4.6), and determining at a given instant which of the 'enabled' handlers are eligible. Let us consider each of these problems in turn.

[1] And perhaps to alter a return address to effect a local transfer of control. See next paragraph.

[2] For example, this location can be at a fixed place in the activation record for the context. Since a pointer to that activation record must be available to the executing handler anyway, local transfer of control is easily effected.

[3] In some implementations, there may be no additional cost in supplying the necessary location.

8.2.1 Enabled Handlers

The enabled handlers set can be partitioned into two subsets: those handlers declared at the module level and those that appear within function bodies. The former are enabled when their associated modules are instantiated, the latter are enabled while control resides within their associated contexts. We will represent these sets by lists. Each list element on these lists will represent an enabled handler and will contain at least:

- a *link* field (to point to the next list element),[4]

- a *context pointer* that identifies the associated context (e.g. points to an activation record - see discussion in section 8.1),

- a *handler pointer* that identifies the code of the handler body, and

- a *unique name.*

The unique name field is, in effect, a token that represents a user, as we shall see shortly.

Now we can distinguish two cases: 'flow' and 'structure' class conditions. Flow conditions constitute a special case and can be easily dismissed. First, module-body level handlers for flow class conditions can always be "distributed" to function-body level, by virtue of the shorthand notation (see section 4.5). This eliminates the need for a module-body level list for flow class conditions. Second, all enabled handlers for a given condition and function instance must appear in the same process stack, by definition. This suggests a simple implementation for all flow class conditions, which is essentially similar to the one used in BLISS. With

[4] There may actually be two link fields in the event that a doubly-linked list is deemed desirable. See subsequent discussion of 'structure' class conditions.

each process, we associate a single, distinguished cell that will act as the list header for flow conditions. Within each process stack, we link all enabled handler list elements together (for 'flow' conditions only!) in stack order. In the unique name field of each list element, we insert a value that uniquely identifies the condition name.[5] As we will see below, it is now a simple matter to determine eligibility for any flow class condition.

'Structure' class conditions are more difficult - both kinds of list are needed to maintain the enabled handlers set. For structural conditions, the headers for these lists appear in the instance to which the handlers are attached. (Recall from section 4.11 that access to these lists is controlled by a mutual exclusion semaphore or similar synchronizing construct.) The unique name field of each list element contains a value that identifies the instance of the module containing the handler.[6] Now the module-body level handlers for a given condition and (data) instance are all linked together in no particular order, and the list is updated whenever a referencing instance (i.e. one containing a handler) with a module-body level handler is created or destroyed. For handlers within function bodies, however, the list maintenance occurs more frequently. The list structure depends to some degree on the number of processes; we will consider a general-case implementation that, under particular language restrictions, could be optimized substantially.

A second list exists for all non-module-body level handlers (for a given data instance and condition). Call this the DCL - Dynamic Context List. When a context is entered that has such a handler, a list element is allocated on the process stack and filled in with the above information. One additional datum is included, a unique value identifying the function instance in which the context exists. (This is in addition to the unique value identifying the module in which it exists. The value of the stack pointer at function entry will serve as a unique identifier.) The element is then linked onto the front of the DCL. Note that the effect is to have several

[5] This is easily and cheaply accomplished by compiler/linker cooperation; no run-time allocation is necessary.

[6] Conceptually, we produce a separate copy of the code every time a module is instantiated. In fact, all we need is a unique number to keep track of instantiations. This number can be generated in any one of several satisfactory ways; a particularly convenient method is suggested in the next section.

separate stacks (one for each process) all "woven" together. When a context having such a handler is exited, the element is unlinked from the DCL and destroyed (popped from the process stack). Note that if the list is singly-linked this will, in general, require a search. However, by adding a second link field and maintaining a doubly-linked list, both entry and exit times become fixed and independent of the interleaving in the list. Of course, entry time becomes slightly greater (2 additional fields to change), but this sacrifice is probably worth the benefit (of fixed, known cost) reaped. The two lists now maintain the enabled handlers set for a structural condition.

8.2.2 Eligible Handlers

Given the enabled handler lists as defined above at some instant, we can determine the eligible handlers set. For flow conditions the task is straight-forward: we simply follow the links back through the current process's enabled handlers list for flow conditions until we find the first element whose unique name field matches the condition being raised. This element is the only eligible handler. Indeed, we can be certain[7] that this list element will appear in the segment of the process stack allocated to the invoker of the signalling function, and if that segment is readily identifiable, we may bound the search. In any case, the algorithm is obvious.

For structure class conditions the task is more complex. The algorithm appears in figure 8.1. SCL stands for 'Static Context List', i.e. the header for the module-body handlers list. DCL, as previously mentioned, stands for 'Dynamic Context List' and is the header for the handlers appearing in function bodies. The relevant fields are:

> *next* - the (forward) link to the next list element,
>
> *mi* - the unique name identifying the module instance containing the handler,

[7] Except in certain pathological cases where a function name is passed as a parameter. Not all languages permit this.

fi - the unique name identifying the function instance containing the handler (meaningless for elements of the SCL), and

mark - a boolean flag (initially false) used internally by the algorithm.

Let us briefly examine the operation of the algorithm. Recall that, by assumption, the SCL and DCL are protected by a mutual exclusion semaphore. Thus the lists cannot be altered during the execution of this algorithm. We will generate the members of the eligible handlers set one-at-a-time in an unspecified order.

The search for eligible handlers extends from lines 1 to 30 and consists of two parts. Lines 3 to 22 locate eligible handlers on the DCL; lines 23 to 29 locate eligible handlers on the SCL. The 'mark' field, if true, indicates that the list element corresponds to a masked handler (in the sense of section 4.6), i.e. an <u>in</u>eligible, *enabled handler.* Initially, all 'mark' fields are false, and the algorithm restores that state upon completion.

The search of the DCL involves finding the lexically innermost handler within each distinct module and function instance. The pointer P is used to examine the elements of the DCL sequentially. The test on line 5 will fail the first time, since all 'mark' fields are presumed false initially. Thus P corresponds to an eligible handler, as the notation on line 8 suggests. (We assume that the handler is initiated at this point and the selection policy indicates whether more handlers are required. See line 33.) We must now eliminate (i.e. mark) all handlers on the DCL that P masks, which is the effect of lines 9 to 13. A little thought reveals that such handlers are completely characterized by the test on line 10 and the fact that they appear "deeper" in the DCL than P does.[8] Having eliminated all such masked handlers, we now must eliminate the single handler on the SCL that P masks. Lines 14 to 18 do this.

On subsequent iterations of the main DCL loop (beginning on line 3), the test

[8] Recall that the DCL is in fact a collection of stacks "woven together".

```
 1 search: begin "search"
 2        P←DCL.next; R←SCL.next;
 3        while P≠nil do
 4          Q←P.next;
 5          if P.mark
 6            then P.mark←false
 7            else
 8              eligible(P);
 9              while Q≠nil do
10                if Q.mi=P.mi and Q.fi=P.fi
11                  then Q.mark←true fi;
12                Q←Q.next
13               od;
14              while R≠nil do
15                if R.mi=P.mi
16                  then R.mark←true; exitloop
17                  else R←R.next
18                 fi od;
19              R←SCL.next
20            fi;
21          P←Q
22         od;
23        while R≠nil do
24          if R.mark
25            then R.mark←false
26            else eligible(R)
27            fi;
28          R←R.next
29         od
30        end "search";
31        while Q≠nil do Q.mark←false; Q←Q.next od;
32        while R≠nil do R.mark←false; R←R.next od
   where:
33  eligible(X) = <invoke X's handler>;
             if no more handlers needed then leave search fi
```

Figure 8.1: Eligibility Determination Algorithm

on line 5 may be *satisfied*. If so, we have encountered a handler previously determined to be masked. In such a case we merely reset its 'mark' field to be false and proceed to the next DCL element. Observe that when execution passes to *line* 23, all elements of the DCL will again have their 'mark' fields set to false and Q will have the value 'nil'.

The search of the SCL on lines 23 to 29 is straight-forward. All marked elements have their 'mark' fields set to false, and all unmarked elements are considered eligible. Note that, by virtue of lines 14 to 18, an element of the SCL can be eligible (unmarked) if and only if no element of the DCL has a matching 'mi' field. This is precisely the definition of eligibility for module-body level handlers, i.e. the handlers corresponding to elements on the SCL. Observe that when execution passes to line 30, all elements of the SCL will have their 'mark' fields set to false and R will have the value 'nil'.

If control reaches line 31 from line 30, lines 31 and 32 have no effect, since Q=R=nil. These loops exist only to "clean up" the DCL and SCL if the selection policy causes premature termination of *the eligibility search*.[9] It is easily verified that if premature termination occurs, all list elements between the head of the DCL (SCL) and Q (R) already have their 'mark' fields set to false. Thus when control passes from line 32, all list elements have been unmarked.

We close this chapter by briefly assessing the cost of this algorithm. At first glance, it seems quite expensive. However, we should observe that the DCL is likely to be quite short, unless there is a high degree of simultaneous access by parallel processes to the same data structure. With a short DCL, the dominant cost of the algorithm becomes the search of the SCL incurred for each unmarked element of the DCL (lines 14 to 18). A clever encoding, however, can eliminate the search entirely. Recall that the 'mi' field is defined above as a unique identifier for the instance of the module containing the handler described by the list element. Since elements of the SCL are in one-to-one correspondence with such instances, the 'mi' field can be taken as the address of the appropriate element on the *SCL!* This address is bound at *module instantiation* time. Now there is no need to search the SCL for the desired element, and we can replace lines 14 to 18 by:

[9] I.e. if the particular selection policy, e.g. sequential-conditional, does not require the entire eligible handlers set.

```
R ← P.mi;
R.mark ← true
```

We now need to examine each element of the SCL only once to determine all its eligible handlers - we couldn't expect to do much better. In pathological cases the processing of a DCL of length 'n' can still require $O(n^2)$ list element accesses, but generally we expect 'n' to be relatively small, particularly compared to the length of the SCL. We also expect masking to reduce the processing time substantially.

Part IV
Conclusion

PRECEDING PAGE BLANK-NOT FILMED

9
Summary

In the preceding chapters we established a set of goals for an exceptional condition handling mechanism, then defined a particular mechanism and verified that it indeed met the goals. We have argued that in meeting these goals, the mechanism significantly advances the state of the art in exception handling. Yet the goals are rather general in character, and we might find it satisfying to assess the mechanism in terms of other, *more specific criteria*. In a sense, such an "independent assessment" confirms that the proposed mechanism represents a useful step forward. This chapter concludes the thesis by providing that assessment and identifying some avenues for future investigation.

9.1 *Contribution of this Work*

To obtain a different perspective on the proposed exception handling mechanism, let us return to the list of issues presented in section 2.1. By examining the behavior of our mechanism with respect to the questions posed there and recalling the observations of chapter 2, we will see clearly the contribution of our mechanism. The sections below respond directly to the questions of section 2.1.

9.1.1 Specification

Our mechanism provides for explicit naming of exceptional conditions. Their semantics are precisely defined by predicates that tell a handler what it can expect and what it must do. These specifications appear in the module that defines the exceptional condition and are exported to those modules that use it. The generality of this specification technique permits the application of the mechanism to a wide range of situations, including some that, under traditional definitions, do not involve exception handling. Our mechanism derives much of its attractiveness and power from this broad applicability, and contrasts sharply with

PRECEDING PAGE BLANK-NOT FILMED

many existing mechanisms that are closely tied to particular language or system constructs.

9.1.2 Abstraction

Our mechanism is centrally concerned with the preservation of abstractions (recall the discussion of section 3.1). By defining the semantics of an exception in terms of the abstract entity provided by a module, the specification technique supports the separation of abstract and concrete objects and distinguishes function from implementation. A common failing of existing mechanisms is their inability to force programmers to use only the abstract properties of a module and ignore its representation. This weakness compromises the integrity of modules by subverting the principle of encapsulation. Our mechanism aids abstraction by placing the definition of exceptions on a par with the definition of functions, making it possible for the user of the abstraction to ignore all representational issues. By forcing programmers to define exceptions in abstract terms, our mechanism also encourages robust module design, since the same precision is required to specify exceptional and normal case behavior.

9.1.3 Sharing

We have dwelt at some length (particularly in section 3.1.3) on the inadequacy of the "calls" hierarchy for exception transmission. It is remarkable that the central notion of shared abstractions has been almost totally ignored by existing exception mechanisms. Indeed, one of the most significant properties of our proposed mechanism is its ability to support shared abstractions in a natural way. Once we realize that sharing is of crucial importance, we see that particular kinds of sharing, e.g. asynchronous, parallel access, can be accommodated as important subcases. Thus we gain the ability to handle exceptions in both sequential and parallel programming environments without introducing special facilities for either one. By recognizing sharing as the fundamental property, parallel sharing "falls out" as a special case. We also elevate a concept long

recognized as important in operating systems to the level it deserves in programming languages.

9.1.4 Programming Flexibility

There is no question that our specification technique forces the programmer to consider the "exceptional case" behavior of his program much more carefully. If he wishes to produce a robust program, the mechanism can aid him in expressing exception processing requirements. However, it may also constrain the modes of expression he can use for related, but non-exceptional case processing. We believe that such restrictions, rather than limiting flexibility, enhance clarity and contribute to correctness, just as disciplined transfers of control and data structuring techniques do. It is true that our mechanism does not mesh well with some familiar programming constructs (such as the non-local goto), and we have no qualms about imposing restrictions on such constructs when they affect program clarity and coherent exception processing. The examples of chapter 7 illustrate that a considerable variety of applications can still be programmed in a natural way while utilizing the facilities of our exception mechanism.

9.1.5 Language

In section 5.1 we examined the relationship between our exception mechanism and its embedding language. Ideally, of course, the addition of an exception handling facility should not perturb the language at all. Our mechanism is largely language-independent and imposes very few restrictions on its embedding language. No specific properties, other than the ability to support abstraction by encapsulation, are required. While several language features, e.g. scope rules, naturally interact with the exception handling mechanism, none is constrained to assume a particular form (in contrast to many existing mechanisms). The most significant effect on the language induced by the exception mechanism appears in the synchronization facilities (if any). Here, the requirements of the mechanism, while defined separately from the language facility, interact with it in non-obvious

ways. Section 5.2 shows how the complexities of this interaction may be reduced and the combination of the two mechanisms used to the programmer's advantage.

9.1.6 Verification

Chapter 6 demonstrates the verifiability of the proposed mechanism within the context of a particular modern programming language. We were careful to define the semantics of the exception mechanism to limit its effect on the verifiability of the host language. Specifically, its semantics were tailored to fit existing program structures, particularly abstract functions. Although the desire for verifiability has affected the fine points of the mechanism, the power of the facility has not been compromised by that desire. The examples of chapter 7, complete with pre- and post-conditions, support that conclusion. While the details of verification will necessarily vary from language to language, the semantic specification is relatively general and should adapt easily to most inductive assertion-based proof methodologies.

9.1.7 Cost

The algorithms presented in chapter 8 show that a cost-effective implementation of the proposed mechanism does exist. Further, the costs are distributed to minimize overhead in the non-exceptional case. We have also seen that if the mechanism is to be used more for communication than exception processing, the costs can be balanced somewhat more equitably between the maintenance of the eligible handlers set and the actual transmission of an exceptional condition. It is difficult to assess the cost-effectiveness of this mechanism with respect to potentially attractive alternatives, since the latter are clearly language-specific. In Alphard there are no obvious contenders. In languages that support procedure variables, it is evident (see section 2.2.4) that for approximately equal overhead cost we get considerably greater flexibility from our proposed mechanism. Most of the other mechanisms of chapter 2 do not provide sufficient function to be considered "attractive alternatives" to our proposal.

9.2 *Remaining Issues*

At a number of points in the preceding chapters, we have identified issues that merit further investigation. In this final section we briefly recapitulate these issues and append a few others that logically follow from this work. We believe that the successful resolution of each of these questions will enhance the mechanism's applicability and hence its value as a program structuring tool.

9.2.1 Selection Policy Primitives

In section 4.7.1 we established principles that selection policies should exhibit and gave three specific illustrations. We can ask whether a language should, in lieu of particular policies, supply primitives that permit programmers to define their own. Such primitives would necessarily be engineered only to produce policies that satisfy the postulated properties. By developing such primitives we can gain insight into the range of interactions between signaller and handlers, and we may identify additional or alternate properties that more precisely bound the space of those interactions.

9.2.2 Verification of Synchronization

In chapter 6 we ignored the difficulties in verification caused by the existence of parallelism within programs that may use the exception mechanism. The field of verification of parallel programs is one of considerable recent research interest, and positive results in this area would enable us to complete the formal specification of our mechanism's semantics. In particular, the synchronization requirements of selection policies could then be precisely expressed, perhaps assisting in the development of additional characteristic properties (see above).

9.2.3 Usage Paradigms

Section 5.2 discusses the problem of combining language primitives in coherent ways. Although the principle of "orthogonal design" espoused by Algol-68 [van Wijngaarden 75] is a laudable one, in practice few languages achieve complete orthogonality. The subtle (and sometimes undesirable) interactions of our exception mechanism with synchronization primitives (see section 5.1.4) is an example. The difficulties can be eliminated, in large part, by supplying a sufficiently rich collection of "composite operations", which respond to commonly occurring programming situations. Section 5.2 illustrates a few such "usage paradigms", but doubtless many others can be found. By accumulating a collection of useful "higher-level" operations, we encourage the use of the exception mechanism, since the composite actions are, in effect, "pre-tested" applications of the mechanism.

9.2.4 Protection

If we consider our mechanism in a system rather than a language context, we become aware of a number of protection issues. In section 5.1.5 we briefly mentioned the possibility that our mechanism contains covert channels for leaking information from a confined domain. There are other questions as well, e.g. should restricting the rights in a capability also restrict the exceptions that may be handled by the environment possessing that capability? The solutions to these questions will enhance the application of the exception mechanism to systems in which protection is a serious concern.

9.2.5 Enforcement

A related issue is enforcement. We rely heavily on a compiler/verifier to ensure that modules meet certain of their specifications. Many operating systems take a more conservative approach and enforce specifications at run-time. In this

way they ensure that no higher-level (in the sense of the "uses" relation) error, inadvertent or deliberate, can result in a violation of specifications. Of course, there is a certain cost incurred by doing all checks dynamically, and for some of the features of the exception mechanism (e.g. ensuring that a handler cannot cause its associated condition to be signalled anew), that cost may be substantial. It is an open question whether our exception mechanism can be implemented with reasonable efficiency in a system that enforces all specifications at run-time.

9.2.6 Hardware Applicability

There are at least two important open questions concerning the interaction of our mechanism and hardware design. First, how can microcode be used to reduce the implementation costs of the mechanism? Is it feasible, for example, to move the eligible handler set maintenance into firmware? Second, what is the effect of the exception mechanism on system architecture? Do we have to revise our view of processor-channel-device communication? Is the mechanism sufficiently general to accommodate sharing at the *instruction or memory cycle* level? Because the ratio of hardware to software cost is small in many modern systems, we may need to re-evaluate the communication schemes we have used to date and incorporate new ones that are more in line with the proposed mechanism. If the goal of uniformity is to be completely achieved, we must explore the possibility of making hardware/software and software/software interfaces consistent.

9.2.7 Uniform Control Structure

There is a constant tension between the desires for pleasing conceptual uniformity and special case efficiency. At several places we have remarked that our proposal can serve as a communication tool, yet we persist in calling it an exception handling mechanism. Is there a more general view of control flow structure that brings exceptional and normal case together in a single, consistent fashion? Of course - *production systems are an example*. Yet we the programming public have not abandoned our conventional Algol-like languages in favor of this

more unified view. This is due in part to the (potential) loss of efficiency in special cases - efficiency we deeply cherish. We stick to programming with a set of constructs each tailored to a particular task precisely because we are unwilling to ignore efficiency to gain uniformity. If a way were known to eliminate the tension between these two fundamental concerns, this thesis would never have been written. As it is, we know of no way to "have our cake and eat it too", so we continue to look for language primitives that supply special case semantics at low cost. There is an obvious avenue of investigation open here; no doubt many language designers believe they are walking down it. Clearly, a complete solution to this problem would make the mechanism we propose obsolete.

9.2.8 Self Applicability

Let us close by looking inward instead of outward. The proposed mechanism implements an abstraction of its own, just as the language in which it is embedded does. Exceptions in the language abstractions may be expressed using the exception mechanism (e.g. inability to perform some language-supplied primitive because of a catastrophic error of some obscure sort). Can failures of the exception mechanism be so expressed? Can we apply the concepts of exception handling explored in this thesis to the mechanism itself? Obviously, the mechanism should be robust, and it can employ various redundancy techniques to check its own operation, yet there is bound to come a point at which the mechanism is forced to admit failure to its users. This circularity leads us to the so-called "hard-core reliability problem", that is, the need to assume that there is a "hard-core" nucleus of software whose operation we trust implicitly. System designers abhor such "hard-cores", because they represent singularities in the system (and because they too often prove to be "soft-cores"!). Must we resign ourselves to trusting the (implementation of the) exception handling mechanism, or can we discover ways to make the mechanism self-applicable? The impact of a solution to this admittedly difficult problem would extend beyond the exception mechanism and cause us to revise our present approach to reliable system design.

Appendix A
A More Flexible Handler Definition

"It's far too clear," he said. "It should be writ in such a way that if you ever changed your mind, the point could be disowned, denied, and disavowed."

- James Thurber, The White Deer

In section 4.9 we defined a syntax that permits a handler to effect a local transfer of control. There are programming situations in which the unconditional posting of a transfer of control (in the terminology of that section) is inconvenient; a more flexible dynamic determination may be preferable. For example, a handler might wish to retry a failing operation three times before ultimately giving up. Obviously, the actions 'retry' and 'give-up' correspond to different control points in the associated context. The syntax of section 4.9 makes such a handler rather awkward to write. In this appendix we suggest a change to the exception mechanism of chapter 4 that permits such a handler to be specified without compromising verifiability.

A more general form of handler specification would permit a completely dynamic determination of the point in the associated context at which execution will resume. First, we relax the requirement that the handler body be a single procedure invocation, permitting instead an arbitrary block. Second, we introduce a language construct 'post l', where 'l' is a label. This construct may appear only within a handler and 'l' must reference a label within the enclosing function body. When 'post l' is executed, 'l' is posted as the location to which control is to be transferred when execution resumes in the associated context.[1] Execution of the handler is then terminated.[2]

[1] Some implementations may wish to restrict 'l' to be a label constant, not a variable.

[2] This property of post is not essential, but it simplifies verification.

It should be evident that 'post l' has the effect of (dynamically) inserting a 'goto l' at the point of interruption of the associated context. The usual proof rule for 'goto l' [Clint 72] requires an assertion to be supplied at 'l', and we make a similar requirement for 'post l'. It is then a simple matter to extend the handler proof rule in section 6.5 to include the proof rule for the goto. We omit the details here, noting only that while the extension is conceptually straightforward, it may induce considerable (albeit mechanical) effort in the verification process.

Appendix B
The Alphard Verification Methodology

[Note: This appendix is taken from [London 76] with slight modifications.]

Alphard's verification methodology is designed to determine whether a <u>form</u> will actually behave as promised by its abstract specifications. The methodology depends on explicitly separating the description of how an object behaves from the code that manipulates the representation in order to achieve that behavior. It is derived from Hoare's technique for showing correctness of data representations [Hoare 72].

The abstract object and its behavior are described in terms of some mathematical entities natural to the problem domain (e.g. graphs, sequences). We appeal to these abstract types

- in the <u>invariant</u>, which explains that an instantiation of the <u>form</u> may be viewed as an object of the abstract type that meets certain restrictions,

- in the <u>initially</u> clause, where a particular abstract object is displayed, and

- in the <u>pre</u> and <u>post</u> conditions for each function, which describe the effect the function has on an abstract object that satisfies the invariant.

The <u>form</u> contains a parallel set of descriptions of the concrete object and how it behaves. In many cases this makes the effect of a function much easier to specify and verify than would the abstract description alone.

Now, although it is useful to distinguish between the behavior we want and the data structures we operate on, we also need to show a relationship that holds

between the two. This is achieved with the representation function rep(x), which gives a mapping from the concrete representation to the abstract description. The purpose of a form verification is to ensure that the two invariants and the rep(x) relation between them are preserved.

In order to verify a form we must therefore prove four things. Two relate to the representation itself and two must be shown for each function. Informally, the four required steps are[1]:

For the form

1. Representation validity
   $I_c(x) \supset I_a(rep(x))$

2. Initialization
   requires { *init clause* } initially $(rep(x)) \wedge I_c(x)$

For each function

3. Concrete operation
   $in(x) \wedge I_c(x)$ { *function body* } $out(x) \wedge I_c(x)$

4. Relation between abstract and concrete
   4a. $I_c(x) \wedge pre(rep(x)) \supset in(x)$
   4b. $I_c(x) \wedge pre(rep(x')) \wedge out(x) \supset post(rep(x))$

Step 1 shows that any legal state of the concrete representation has a corresponding abstract object (the converse is deducible from the other steps). Step 2 shows that the initial state created by the representation section is legal. Step 3 is the standard verification formula for the concrete operation as a simple

[1] We will use $I_a(rep(x))$ to denote the abstract invariant of an object whose concrete representation is x, $I_c(x)$ to denote the corresponding concrete invariant, italics to refer to code segments, and the names of specification clauses and assertions to refer to those formulas. In step 4b, "pre(rep(x'))" refers to the value of x before execution of the function. A complete development of the form verification methodology appears in [Wulf 76a, Wulf 76b].

program; note that it enforces the preservation of I_c. Step 4 guarantees (a) that the concrete operation is applicable whenever the abstract pre condition holds and (b) that if the operation is performed, the result corresponds properly to the abstract specifications.

References

"You'd just get mad. It's full of clauses and phrases and pauses, and marginal notes and inner quotes, and words in Latin and words in Greek, viz. *and* ibid. *and* circa *and* sic.*"*

- James Thurber, The White Deer

[Baker 72] Baker, F., Chief Programmer Team Management of Production Programming. *IBM Systems Journal* (1972), pp. 56-73.

[Brinch Hansen 71] Brinch Hansen, P., RC4000 Software Multiprogramming System, Second edition, A/S Regnecentralen, Copenhagen, 1971.

[Brinch Hansen 72] Brinch Hansen, P., Structured Multiprogramming. *Communications of the ACM* 15, 7 (July 1972), pp. 574-578.

[Bron 76] Bron, C. et. al., A Proposal for Dealing with Abnormal Termination of Programs, Twente University of Technology, Technical Report, 1976.

[Campbell 74] Campbell, R. H. and Habermann, A. N., The Specification of Process Synchronization by Path Expressions. *Lecture Notes in Computer Science*, vol. 16, Springer-Verlag, Berlin (1974).

[Clint 72] Clint, M. and Hoare, C. A. R., Program Proving: Jumps and Functions. *Acta Informatica* 1 (1972), pp. 214-224.

[Cohen 75] Cohen, E. and Jefferson, D., Protection in the Hydra Operating System. Proceedings of the Fifth Symposium on Operating Systems Principles, *Operating Systems Review* 9, 5 (1975), pp. 141-160.

[DEC 74] Digital Equipment Corporation, BLISS-11 Programmer's Manual, Maynard, Mass., 1974.

PRECEDING PAGE BLANK-NOT FILMED

[Dijkstra 68] Dijkstra, E. W., Co-operating Sequential Processes. In Programming Languages, F. Genuys (ed.), Academic Press, New York, 1968.

[Dijkstra 76] Dijkstra, E. W., A Discipline of Programming, Prentice-Hall, Englewood Cliffs, N.J., 1976.

[Geschke 77] Geschke, C. M. et. al., Early Experience with MESA. Proceedings of an ACM Conference on Language Design for Reliable Software, *SIGPLAN Notices 12*, 3 (1977).

[Goodenough 75] Goodenough, J. B., Exception Handling: Issues and a Proposed Notation. *Communications of the ACM 18*, 12 (Dec. 1975), pp. 683-696.

[Hibbard 76] Hibbard, P., Parallel Processing Facilities. In *New Directions in Algorithmic Languages - 1976*, S. A. Schuman (ed.), IRIA, 1976, pp. 1-7.

[Hoare 69] Hoare, C. A. R., An Axiomatic Basis for Computer Programming. *Communications of the ACM 12*, 10 (Oct. 1969), pp. 576-580.

[Hoare 72] Hoare, C. A. R., Proof of Correctness of Data Representations. *Acta Informatica 1*, 4 (1972), pp. 271-281.

[Hoare 74] Hoare, C. A. R., Monitors: An Operating System Structuring Concept. *Communications of the ACM 17*, 10 (Oct. 1974), pp. 549-557.

[Horning 74] Horning, J. J., A Program Structure for Error Detection and Recovery. *Proc. Conf. on Operating Systems: Theoretical and Practical Aspects*, IRIA (1974).

[IBM 70] IBM Corporation, PL/I (F) Language Reference Manual, Form GC28-8201, IBM Corporation, 1970.

[Lampson 69] Lampson, B. W., Dynamic Protection Structures. *AFIPS Conference Proceedings*, FJCC (1969).

[Lampson 74] Lampson, B. W., Mitchell, J. G., and Satterthwaite, E. H., On the Transfer of Control Between Contexts. In *Lecture Notes in Computer Science*, vol. 19, B. Robinet (ed.), Springer-Verlag, N.Y., 1974, pp. 181-203.

[Lampson 77] Lampson, B. W. et. al., Report on the Programming Language EUCLID. *SIGPLAN Notices 12, 2* (Feb. 1977).

[London 76] London, R., Shaw, M., and Wulf, W., Abstraction and Verification in Alphard: A Symbol Table Example, Carnegie-Mellon Univeristy Department of Computer Science Report, 1976.

[Parnas 72a] Parnas, D. L., Information Distribution Aspects of Design Methodology. *Proc. IFIP Congress*, vol. 1 (1972).

[Parnas 72b] Parnas, D. L., A Technique for Software Module Specification. *Communications of the ACM 15*, 5 (May 1972), pp. 330-336.

[Parnas 72c] Parnas, D. L., Response to Detected Errors in Well-structured Programs, Carnegie-Mellon University Department of Computer Science Report, 1972.

[Parnas 72d] Parnas, D. L., On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM 15*, 12 (Dec. 1972).

[Parnas 74] Parnas, D. L., On a Buzzword: Hierarchical Structure. *Proc. IFIP Congress*, North-Holland Publ. Co. (1974).

[Parnas 76] Parnas, D. L. and Wurges, H., Response to Undesired Events in Software Systems, T. H. Darmstadt, 1976.

[Pollack 77] Pollack, F. J., A Design Methodology for Fault-Tolerant Software, Ph. D. thesis, Carnegie-Mellon University, 1977.

[Randell 75] Randell, B., System Structure for Fault Tolerance. Proceedings of the International Conference on Reliable Software, *SIGPLAN Notices 10*, 6 (Jun. 1975), pp. 437-449.

[Ross 67] Ross, D. T., The AED Free Storage Package. *Communications of the ACM 10*, 8 (Aug. 1967), pp. 481-492.

[Rychener 76] Rychener, M. D., Production Systems as a Programming Language for Artificial Intelligence Applications, Ph. D. thesis, Carnegie-Mellon Univeristy, 1976.

[Schroeder 72] Schroeder, M., Cooperation of Mutually Suspicious Subsystems, Ph. D. thesis, Massachusetts Institute of Technology, MAC-TR-104, 1972.

[Wasserman 77] Wasserman, A. I., Procedure-Oriented Exception Handling, Univ. of Calif. S.F., Laboratory of Medical Science Report, 1977.

[van Wijngaarden 69] van Wijngaarden, A. (ed.), Report on the Algorithmic Language ALGOL 68. *Numerische Mathematik 14*, 2 (1969), pp. 79-218.

[van Wijngaarden 75] van Wijngaarden, A. et. al., Revised Report on the Algorithmic Language ALGOL 68. *Acta Informatica* v. 5, Fasc. 1-3 (1975).

[Wirth 66] Wirth, N. and Hoare, C. A. R., A Contribution to the Development of ALGOL. *Communications of the ACM* 9, 6 (Jun. 1966), pp. 413-431.

[Wulf 74] Wulf, W. A. et. al., Hydra: The Kernel of a Multiprocessor Operating System. *Communications of the ACM 17*, 6 (Jun. 1974), pp. 337-345.

[Wulf 76a] Wulf, W. A., London, R. L., and Shaw, M., Abstraction and Verification in Alphard. In *New Directions in Programming Languages - 1975*, S. A. Schuman (ed.), IRIA, 1976.

[Wulf 76b] Wulf, W. A., London, R. L., and Shaw, M., Abstraction and Verification in Alphard: Introduction to Language and Methodology, Carnegie-Mellon University and USC Information Sciences Institute Technical Reports, 1976.

[Wulf 76c] Wulf, W. A., London, R. L., and Shaw, M., An Introduction to the Construction and Verification of Alphard Programs. *IEEE Transactions on Software Engineering*, SE-2, 4 (Dec. 1976), pp. 253-265.

[Wulf 78] Wulf, W. A. and Levin, R., The Hydra Operating System, (Monograph to appear), 1978.