2

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188



**1. AGENCY USE ONLY (Leave blank)**

**2. REPORT DATE**
January 1990

**3. REPORT TYPE AND DATES COVERED** Reprint
Technical Report, UMIACS-TR-90-2

**4. TITLE AND SUBTITLE**
DATA-ORIENTED EXCEPTION HANDLING

**5. FUNDING NUMBERS**
AFOSR-87-0130
61102F 2304/A6

**6. AUTHOR(S)**
Qian Cui

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Department of Computer Science
University of Maryland
College Park, MD 20742

**8. PERFORMING ORGANIZATION REPORT NUMBER**
AFOSR-TR- 91 0107

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
AFOSR/NM
Bldg 410
Bolling AFB DC 20332-6448



**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
AFOSR-87-0130

**11. SUPPLEMENTARY NOTES**

**12a. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release;
distribution unlimited.

**12b. DISTRIBUTION CODE**

**13. ABSTRACT (Maximum 200 words)**

mechanisms were added to programming languages to segregate algo- ...rror processing. However, there is no consensus on how to define ...aching handlers to control statements clutters source text in much ...arameters for suitability as inputs for an operation and significance ...issertation, we present a definition for exceptions and a set of ... our definition by associating exceptions with the operations of a ...bjects. We call our notation *data-oriented exception handling* to ...ontrol-oriented versions. We describe the implementation of a ... to Ada. Case studies of programs indicate that control- ...anisms are poorly understood and used. Experimental results ...on handling can be used to produce programs that are small- ...o understand and modify. With the exception of pre- ... space penalty is incurred using data-oriented exception

...ms written in our notation, we extend the proof rules for ...test coverage metrics to assess how well test data ex- ... handlers. Comparisons of proofs of programs with ...how that those for data-oriented exception handling ... statements for existing exceptions or new excep- ... coverage are also simpler for data-oriented than

**14. SUBJECT TERMS**

**15. NUMBER OF PAGES**

**16. PRICE CODE**

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | SAR |

NSN 7540-01-280-5500

91 3 06 181

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18

## GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

**Block 1.** Agency Use Only *(Leave blank)*.

**Block 2.** Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

**Block 3.** Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

**Block 4.** Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

**Block 5.** Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

| | | | |
|---|---|---|---|
| C | - Contract | PR | - Project |
| G | - Grant | TA | - Task |
| PE | - Program Element | WU | - Work Unit Accession No. |

**Block 6.** Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

**Block 7.** Performing Organization Name(s) and Address(es). Self-explanatory.

**Block 8.** Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

**Block 9.** Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

**Block 10.** Sponsoring/Monitoring Agency Report Number. *(If known)*

**Block 11.** Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

**Block 12a.** Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- DOE - See authorities.
- NASA - See Handbook NHB 2200.2.
- NTIS - Leave blank.

**Block 12b.** Distribution Code.

- DOD - Leave blank.
- DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- NASA - Leave blank.
- NTIS - Leave blank.

**Block 13.** Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

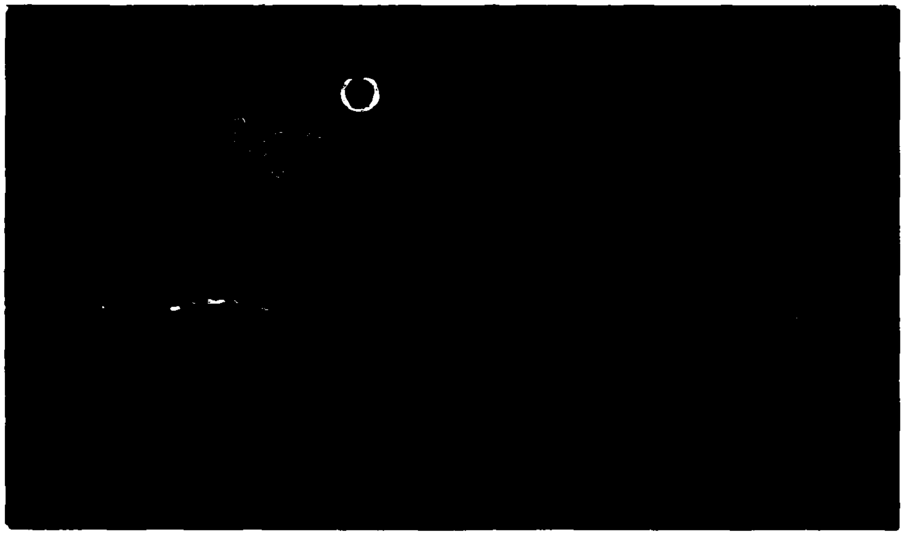
**Block 14.** Subject Terms. Keywords or phrases identifying major subjects in the report.

**Block 15.** Number of Pages. Enter the total number of pages.

**Block 16.** Price Code. Enter appropriate price code *(NTIS only)*.

**Blocks 17. - 19.** Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

**Block 20.** Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.



# COMPUTER SCIENCE
# TECHNICAL REPORT SERIES

## UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

UMIACS–TR–90–2
CS–TR–2384
January 1990

**Data–Oriented Exception Handling**

Qian Cui
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

Exception handling mechanisms were added to programming languages to segregate algorithmic processing from error processing. However, there is no consensus on how to define exceptions. In addition, attaching handlers to control statements clutters source text in much the same way that testing parameters for suitability as inputs for an operation and significance as results does. In this dissertation, we present a definition for exceptions and a set of language features that support our definition by associating exceptions with the operations of a type and handlers with data objects. We call our notation *data-oriented exception handling* to distinguish it from the usual control-oriented versions. We describe the implementation of a pre-processor from our notation to Ada. Case studies of programs indicate that control-oriented exception handling mechanisms are poorly understood and used. Experimental results indicate that data-oriented exception handling can be used to produce programs that are smaller, better structured, and easier to understand and modify. With the exception of pre-processing time, no significant time or space penalty is incurred using data-oriented exception handling.

In order to verify and test programs written in our notation, we extend the proof rules for several Ada constructs and develop new test coverage metrics to assess how well test data exercises bindings of raise statements and handlers. Comparisons of proofs of programs with different exception handling approaches show that those for data-oriented exception handling require less change in response to new raise statements for existing exceptions or new exception declarations. Algorithms to assess test coverage are also simpler for data-oriented than control-oriented mechanisms.

# Data-Oriented Exception Handling

by

Qian Cui

Dissertation submitted to the Faculty of the Graduate School

of The University of Maryland in partial fulfillment

of the requirement for the degree of

Doctor of Philosophy

1989

Advisory Committee:

Professor John Gannon, Chairman/Advisor

Associate Professor Marvin Zelkowitz

Associate Professor Alan Hevner

Assistant Professor Dieter Rombach

Assistant Professor Leo Mark

## Acknowledgments

I am deeply indebted to Dr. John Gannon, my advisor, for his invaluable guidance and support. Without his untiring interest in discussions and patience in reading endless drafts, this dissertation would have remained only as an idea. I would also like to thank Dr. Marvin Zelkowitz, Dr. Alan Hevner, Dr. Dieter Rombach, and Dr. Leo Mark for participating in my Committee and providing useful comments.

I also wish to thank my students of the advanced Ada course in the University of Maryland, University College for taking part in the experiments related to this work.

Last but not least, I am grateful to my wife Weina for her continuing encouragement and unfaltering support.

The support of the Air Force Office of Scientific Research under grant AFOSR-87-00130 is gratefully acknowledged.

# Table of Contents

Chapter 1: Introduction ........ 1

Chapter 2: Survey of Previous Work ........ 12

Chapter 3: Problems with Existing Mechanisms ........ 28

Chapter 4: The New Mechanism ........ 39

Chapter 5: Implementation ........ 51

Chapter 6: Program Verification ........ 65

Chapter 7: Testing ........ 77

Chapter 8: Empirical Studies ........ 101

Chapter 9: Conclusion ........ 111

Appendix: Experimental Studies ........ 115

References ........ 130

# CHAPTER 1

# Introduction

Programming languages evolve with our improved understanding of programming practices. New features are added to languages to provide linguistic support for new design methods, while some old features die out when they become obstacles to contemporary programming practices. Procedures were introduced to permit programmers to refer to common code by name rather than duplicating it. Stepwise refinement further encouraged the use of procedures to facilitate functional abstraction. Packages or modules were added to languages to support information hiding or data abstraction. Similarly, exception handling mechanisms were adopted to segregate error handling code from code implementing algorithms.

It has been about twenty–five years since the first attempt at incorporating exception handling mechanisms into programming languages[PL/I 76]. Although this time period is relatively long compared to the short history of programming languages, few widely used languages have incorporated these features. With increasing use of Ada[Ada 82], it is now possible to investigate the use of its exception handling features in application programs and assess their impact on programming practices.

We define exceptions only in response to implementation insufficiencies [Black 83], which generally occur when the storage reserved for an object is inadequate to represent its value or when performance constraints cannot be met. Although defining resource requirements in an operation's specification might permit programmers to test for such conditions explicitly before invoking an operation, such requirements complicate specifications. Our view of exceptional conditions as implementation insufficiencies results in an exception handling mechanism that is tightly coupled with Ada's package construct implementing abstract data types. Exceptions are defined and raised only in packages because such conditions are defined in terms of an object's representation, which can be manipulated only in a package body. Each data object declared has its own set of (exception, handler) binding pairs specified in its declaration so users can choose different responses to exceptional conditions. Attaching handlers to control statements clutters source text in much the same way that testing suitability of inputs for an operation and the significance of its results does. In contrast, associating handlers with the declarations of types and objects separates centralizes information about exceptional processing away from algorithmic processing. Empirical results indicate that data-oriented exception handling can be used to produce programs that are smaller, better structured, and easier to understand and modify. With the exception of pre-processing time, no significant time or space penalty results from this change.

## 1.1. What is an Exception?

Although the terms *exception* and *exception handling* have been used for quite some time, no rigorous definition for them is accepted. In [Goodenough 75], *exception conditions* are defined as those that are brought to the attention of the operation's invoker. These conditions can be errors like *domain failure* or *range failure*, or classifications of the result of an operation. An exception condition can even be used to monitor an operation, making it another communication link between an invoker and the called operation.

The designers of the exception handling mechanism in CLU[Liskov 79] also defined an exception generally as an unusual occurrence. However, if the exception handling mechanism were efficient enough, exceptions might also be used to convey information about normal situations. Exception handling mechanisms communicate information among procedures at different levels. While such communication can be used to recover from faults such as erroneous data and failures of lower level modules, it can also be used for other purposes.

Levin[Levin 77] even avoided giving a definition of "exception" because he was afraid that doing so might limit the applicability of his proposed mechanism. He preferred to include "errors" as a proper subset of exceptions and proposed using his mechanism for inter-process communication.

In Ada[Ichbiah 79], exceptions are "errors or other exceptional situations that arise during program execution." The Ada Language Reference Manual (LRM) classifies errors into the following four categories: errors that must be detected at compilation time by every Ada compiler; errors that must be detected at run time by the execution of an Ada program; erroneous execution; and incorrect order dependencies. For the second case, the LRM further explains that "the corresponding error situations are associated with the name of the predefined exceptions." Thus the LRM effectively defines exceptions in terms of errors which are in turn defined in term of exceptions. As to the "exceptional situations," the LRM does not give any further definition. Therefore, a programmer has the freedom of declaring any event to be an exception.

In summary, it seems that exceptions cannot be defined except in terms of "errors," "exceptional cases," "rare situations," or "unusual events," which are themselves illdefined terms. Giving a rigorous definition for "exception" requires separation of "normal" and "exceptional" cases. The subjectivity of this distinction can be seen by considering the exceptions associated with symbol table operations. When looking up an item in a symbol table, it may be "exceptional" if the item is not in the table. Thus, **key_not_found** is an exception associated with the look-up operation. However, when an insertion operation is implemented using the look-up operation, exceptional cases of **key_not_found** suddenly become the expected cases. Thus, the event **key_not_found** can be either normal or exceptional depending on one's viewpoint. Since an exception can be anything one likes, exception handling mechanisms have been designed that are general enough to take the role of the procedure calls, multiple exits from procedures, or even interprocess communication mechanisms.

To have a precise and reasonable definition of the term "exception," we need to eliminate events that are not exceptions. An exception should not be an unanticipated program condition. Exception handling mechanisms deal only with well-specified, expected situations. Detecting an unexpected program state is difficult enough without trying to decide how to repair it. An exception should not be a programming error. If an implementation does not conform to its specification, it contains errors. These errors must be *corrected* rather than *handled* in order for the program to function correctly. We cannot use an exception handling mechanism to debug a program because the mechanism is not designed to correct unexpected errors. On the other hand, if a program raises an exception described in the specification, the responses are also defined in the specification. With this point in mind, the inclusion of the exception condition and its handler is not an error, but the accurate implementation of the specification.

An exception should not be a "domain failure." The invoker of an operation must make sure the operation's input assertion is satisfied when invoking the operation. If not, the operation should not be invoked at all. When a partial function is called with an input that lies outside its domain, there is no way for such a function to "fix" the erroneous input. The function can only report its undesired usage, and either return an arbitrary value or abandon program execution. If the invoker is careful enough to provide a handler for such a failure, he could just as easily test the input assertion first to avoid unnecessary computation by the function. On the other hand, if the invoker is unaware of the possible domain failure, then raising an exception will not help since the invoker will not have provided a handler for the exception.

An exception should not be a "range failure." If a function is implemented correctly and if its input assertion is satisfied when the function is called, the function should produce an output satisfying the output assertion. Otherwise, the implementation does not agree with its specification, and errors exist in the implementation. Raising an exception in response to a range error is unlikely to help the invoker since he has no idea how the function is implemented.

Instead of being defined with respect to programming blunders, exceptions should only be defined for situations where a function or operation is logically correct but, due to some limitation imposed by the underlying system, could not be computed. Black[Black 83] calls these situations "*implementation insufficiencies.*" For example, an overflow resulting from an addition operation is an exception because it is caused by insufficient hardware resources (i.e., the word length is too small). Similarly, operations failing to meet performance goals can be defined as implementation insufficiencies. In contrast, a stack underflow when invoking a **pop** operation is a domain failure rather than an exception because the operation is only defined for non-empty stacks. Although the problems of implementation insufficiencies could be partially resolved by defining the resource requirements for an operation in its specification, this approach may not be desirable because this detail clutters specifications by combining descriptions of size and function together.

Conceptually, there is nothing wrong with a module when it fails to perform due to a resource shortage. If more resources are obtained, the module can meet its specification. There is no software error to be corrected in this case. An implementation insufficiency is different from a "range failure." For example, if a **push** operation fails to produce an enlarged stack due to insufficient pre-allocated memory space, the result should still be thought of as falling into the range of the push operation since its range is the set of all stacks (without regard to their sizes).

## 1.2. Components of an Exception Handling Mechanism

An exception handling mechanism allows a problem solution to be divided into normal and exceptional computations and isolates the cases from each other. The language features supporting exception handling can be divided into a set of components: declaring exceptions, binding handlers to exceptions, and raising exceptions. Among the language design issues that arise for these components are how to provide information about the environment to the handler, where control should resume after an exception has been handled, and what happens if an exception is not handled.

### 1.2.1. Declaring Exceptions

In most existing exception handling mechanisms, there are usually two types of exceptions: pre–defined and user–defined. Pre–defined exceptions are declared implicitly and associated with conditions that can be detected by a language's run–time system. The most common pre–defined exceptions are: numeric overflow, array subscript bound error, storage error, etc. Pre–defined exceptions permit programmers to monitor and respond to these conditions should they arise.

User–defined exceptions permit a programmer to declare conditions as exceptions by associating identifiers with the conditions. These conditions are often defined in terms of some application domain at a higher level of abstraction. For example, an exception `stack_overflow` can be declared as part of the specification of a user defined data type `stack_type`.

### 1.2.2. Raising Exceptions

When the conditions associated with some exceptions arise during program execution, these exceptions are brought to the attention of the exception handling mechanism. For the pre–defined exceptions, detection and notification are usually performed automatically by the run–time system. For user–defined exceptions, programmers write code to test conditions and notify the handlers (generally with a `raise` or `signal` statement).

### 1.2.3. Binding Handlers to Exceptions

Corrective actions can be associated with exceptions in a program. Once a specific exception is raised, an action associated with the exception (if any) is located and executed. This process is called *handling exceptions* and the code executed in correspondence to the raised exception is called an *exception handler.*

Handlers may be associated with exceptions dynamically or statically. For dynamic association, a statement must be executed binding a handler to an exception (e.g., the PL/I **ON** statement). The binding remains in effect until another such statement is executed or the scope unit containing the statement terminates. Thus it is impossible for the compiler to determine whether there is a handler associated with an exception at an arbitrary point in the program. Static association is made by tagging a program unit (e.g., block, statement, expression) with a handler which remains in effect while the unit executes.

### 1.2.4. Propagating Exceptions

Exceptions are not always handled successfully. There may not be a handler bound to the exception or the handler for the exception may not be able to repair the exception satisfactorily. The same exception can be raised again in the invoker to search for a handler, until either a handler is found or control passes out of the highest level of procedure invocation. This process is called *propagation.* Since an exception can propagate outside its scope, an exception handling mechanism must provide a way to recognize such an exception. Some mechanisms require that unhandled exceptions be converted to a pre-defined, global exception before being propagated upward. Although propagation makes an exception handling mechanism seem more flexible and powerful, once an exception is propagated outside its original environment much useful context information is lost.

### 1.2.5. Transferring Control after Exception Handling

There are two basic models for transferring control after a handler executes: termination and resumption. In the termination model, the program unit that raised the exception is terminated and control transferred to the statement following the unit's invocation. In the resumption model, control returns to the point following the statement raising the exception. Care must be taken that the condition causing the exception to be raised does not still exist or unpredictable results may be obtained.

The termination model is generally simpler than the resumption model because it is easier to restore a well-defined program state (e.g., the caller's state saved when a procedure was invoked). However, the resumption model often provides useful functions. For example, in adding a list of numbers if numeric overflow occurs, switching to a number representation with a wider range to hold the intermediate results permits the summation to continue. With the termination model, once the overflow occurs, all computations up to that point have to be abandoned and restarted.

### 1.2.6. Passing Parameters

Some local context useful in diagnosis and treatment of exceptions can be transmitted to handlers via parameters. A `raise` statement supplies the actual parameters for an exception, and the corresponding handler uses formal parameters to access the information passed to it. Few exception handling mechanisms permit exceptions to have parameters. As a result, global variables are often used to transmit information, increasing module coupling and program complexity.

## 1.3. Organization of the Dissertation

In this dissertation, we define exceptions as implementation insufficiencies, and propose a new exception handling mechanism suitable for solving such problems. In our mechanism, exceptions are defined on data types because implementation insufficiencies are signaled by operations of the types; handlers are associated with exceptions in object declarations so that users may specify different repair actions for different objects when implementation insufficiencies arise on these objects. By analyzing programs with different exception handling methods, evaluating the impact of our mechanism on program verification and validation, and comparing the performance of programmers as they construct, study, and modify programs, we show that our view of exceptions and new mechanism improve program quality.

Chapter 2 surveys several existing and proposed exception handling mechanisms. Chapter 3 discusses problems inherent in conventional exception handling mechanisms and presents an analysis of exception handling from Ada programs in the Simtel20 Ada Repository.

In Chapter 4, a new exception handling mechanism associating exceptions with a type's operations and binding handlers to exceptions in the declaration of data objects is proposed and demonstrated. Chapter 5 describes the implementation of the proposed mechanism in Ada.

Chapter 6 and 7 consider issues of program verification and validation. A set of proof rules is formulated to prove the correctness of Ada programs employing the proposed mechanism. The simplicity of our method becomes obvious when it is compared with the methods used for Ada's exception handling mechanism. A simple test coverage metric is introduced. By comparing the number of test cases needed for testing a pair of programs employing different exception handling methods, we can see that data-oriented exception handling can help reduce the effort expended in program testing. We also discuss the implementation of a pre-processor to assess test coverage automatically.

Chapter 8 examines the results of some experimental studies which reveal how different exception handling mechanisms impact programmers as they build, study, and modify programs. Chapter 9 summarizes this work and concludes the dissertation by considering possible future research directions.

# CHAPTER 2

# Survey of Previous Work

This chapter briefly surveys innovative exception handling mechanisms that have been proposed or implemented.

## 2.1. Exception Handling in PL/I

PL/I was the first general purpose programming language to include facilities for handling exceptions[PL/I 76] [MacLaren 77]. In PL/I's terms, these exceptions are called *conditions* which are either predefined by the system (e.g., `ENDFILE`, `ZERODIVIDE`, etc.) or declared in a program. When a condition is raised either implicitly or explicitly via a `SIGNAL` statement, the execution of the program is interrupted and control is transferred to the most recently established handler for that condition. If there is no handler associated with an exception when it is raised, a default action is taken.

A handler is associated with an exception by executing an `ON` statement, which binds the handler to the condition named by the `ON` statement and deactivates the previous association for that condition. The newly established association remains in effect until the end of its enclosing block is reached; at that time the handler for the dynamically enclosing block (if any) again becomes active.

When a handler terminates (either normally or by executing an `END` statement,) the signaler's execution is resumed. If resumption is undesirable or prohibited (e.g., for some language-defined conditions), a handler can execute a `STOP` statement to terminate the entire program or use a GOTO statement to transfer control to any place in the program. A raised exception will be propagated to the current block's dynamic enclosing block (if any) if there is no handler associated with it in the current block (i.e., no ON statement binding a handler to the exception has been executed in the current block).

Each language-defined condition has a default handler, which is invoked if no user-defined handler is established. However, default handlers do not treat exceptions uniformly; some default actions abort the program, while others resume the interrupted execution. For example, the UNDERFLOW condition for a floating point operation has a default handler that prints an error message and the returns with zero as the evaluation result. Conversely, the handler for FIXEDOVERFLOW condition is not allowed to return.

Most language-defined conditions do not take parameters, and user-defined conditions are not allowed to take parameters. Thus, communication between a signaler and its handler can only be achieved through global variables. Special cases are the file conditions (e.g., ENDFILE and ENDPAGE), which pass file parameters to their handlers.

### 2.2. Goodenough's Mechanism

The first structured exception handling mechanism was proposed by Goodenough[Goodenough 75]. He argued that exceptions should be declared to be one of three types in order to specify explicitly their resumption or termination constraints. ESCAPE exceptions require termination of the operation raising the exception, NOTIFY exceptions forbid termination, and SIGNAL exceptions may choose either termination or resumption. An exception must be declared to be one of the three types and must be raised by matching statements.

Handlers are associated with exceptions statically by attaching handlers to the end of an executable program unit (e.g., expression, statement, or block), e.g.,

```
IF ( A + B > C ) [ Overflow : EXIT(TRUE); ] THEN ...

CALL P(A); [ ExceptionRaisedByP : CALL F;
                 [ ExceptionRaisedByF : ESCAPE E3; ] ]
Sum = 0;
CALL Scan( P, V ); [ Value : Sum = Sum + V; RESUME; ]
```

The scope of a handler is the same as the scope of the fragment to which it is attached. If a raised exception lies within the scope of a handler for that exception, the handler is executed; otherwise, the same exception is raised within the subroutine's invoker.

If an exception is of type ESCAPE or SIGNAL, the handler can terminate the operation raising the exception by executing an EXIT statement, or by raising an ESCAPE-type exception. Executing a RETURN statement causes the handler to exit and returns control to the invoker of the subroutine containing the handler. For an exception of type NOTIFY or SIGNAL, the handler can resume the operation raising the exception by executing the RESUME statement.

There are some system-defined exceptions like ENDED and CLEANUP. The system supports declaring default exceptions and default handlers. Since exceptions do not take parameters, any communication between an operation raising an exception and the corresponding handler must be performed through global variables.

### 2.3. Levin's Exception Handling Mechanism

In [Levin 77], exceptions are divided into two classes: *structure-class* conditions and *flow-class* conditions. A structure-class condition is raised relative to a data instance, and may impact all users of the data instance. In contrast, a flow-class exception is raised relative to the invocation of an operation and is only interesting to the invoker. As an example, a module implementing a file abstraction may specify (among others) two exceptions: **file-inconsistent** and **file-read-only**. When a user attempts to write to a read-only file, **file-read-only** will be raised. In this case, only the invoker is responsible for the exception raised and a handler within the caller (if an appropriate one is found) is executed. If **file-inconsistent** is raised (when two users write to a file without either gaining mutually exclusive control), all the users of that file are notified to handle the exception. To facilitate communication between the signaler and the handler(s), exceptions may take parameters.

The declaration of an exception does not explicitly specify its class as a structure or flow condition. Such a distinction is made in a **raises** clauses attached to the heading of the exception's signaler, e.g.,

```
condition file-inconsistent
condition file-read-only

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

Since **file-inconsistent** is raised on an object (a file), it must be a structure class condition. In contrast, **file-read-only** is raised on a function invocation and is therefore a flow-class condition.

A handler is associated with an exception statically by attaching the handler to the end of an executable program unit (e.g., statement, block, or function body). Several handlers may be *eligible* for execution if a structure-class exception is raised, thus *selection policies* are formulated to choose one or more of them for execution. For example, if a `storage-pool-low` exception is raised, any process using the object on which the exception was raised can handle the exception by releasing some storage it holds. To coordinate processing among the processes when such an exception is raised, Levin suggests three selection policies: *broadcast-and-wait*, in which the exception is raised in every process eligible and the signaler waits until all handler operations are completed; *broadcast*, in which the signaler does not wait; and *sequential*, in which the handlers are executed one by one but whenever the exception condition is handled the remaining handlers are not executed. Thus, handling structure-class exceptions is actually an inter-process communication and resource management problem. Levin's mechanism does not provide an explicit rule concerning unhandled exceptions. It is implied that this might be a programming error[Levin 77].

Levin's exception handling mechanism forbids a handler from terminating the execution of an operation raising the exception. The signaler's execution always continues immediately following the `raise` statement after the handler finishes execution. Thus, a handler is not allowed to execute a statement like `exit` or `return` to abort the signaler. Levin argues that this is necessary for ensuring that an abstraction raising an exception will always be in a consistent state.

Although a handler cannot alter the flow of control in the signaler, it can change the local flow of control within its associated context. This is intended to cope with problems where control should not return to the point following the invocation if an exception is raised. Levin thus introduces a special syntax form:

$$\textit{statement} \; [\; \textit{condition} : \textit{handler} \rightarrow \textit{control\_transfer} \;]$$

into his mechanism for that purpose. Consider the following example:

```
S1;                                      condition cond-1
L: begin                                 function P2 ( ... )
     ...                                 begin
       P2 [cond-1: H1 → leave L];            ...
       S3;                                   raise cond-1;
       ...                                   S5;
   end                                       ...
S4;                                      end
```

When cond-1 is raised in P2, the associated handler (i.e., H1) is executed. After H1 finishes execution, P2's execution is resumed at S5. However, upon P2's termination, control will not transfer to S3; rather, the whole block labeled by L is exited and S4 is the next statement to be executed.

### 2.4. Exception Handling in CLU

CLU[Liskov 79] uses a single-level termination model. Raising an exception terminates the signaling procedure, and the exception can only be handled by the immediate caller. Thus, instead of a single return path, each procedure has several return paths. One of these is considered the normal path, while others are considered exceptional.

In a procedure heading, a list of exceptions (which may take parameters) can be declared. These are the exceptions that can be raised by the procedure. CLU has one language-defined exception, named **failure**, which may be signaled by every procedure. **failure** is implicitly declared for every procedure and need not be listed in the procedure heading explicitly. If an invoker does not supply a handler for an exception raised in the invocation, that exception is automatically converted to the exception **failure** and the invoker itself is terminated. **failure** takes a string parameter explaining the reason for the exception being raised.

In CLU, handlers are statically associated with invocations and handlers may be attached only to statements. An exception raised within a handler body causes the procedure or block containing the handler to terminate. Thus, there is no risk of recursively raising an exception in its handler. The handler body may also be terminated by an **exit** statement, which is another way to raise exceptions. The difference between a **signal** statement and an **exit** statement is that the former activates a handler in the calling procedure invocation, while the latter activates a handler in the current procedure invocation. In the following example,

```
test = proc(x, y : int) returns(int) signals(bad_num(int))
    ...
    begin      % beginning of a block
        A := sign( x ) except when neg( I : int ) :
                                 S1
                                 signal bad_num(x)
                       end
        B := sign( y ) except when neg( I : int ) :
                                 S2
                                 exit done
                       end
        ...
    end except when done : ... end
    ...
end test
```

the **signal** statement terminates execution of the procedure **test** and returns to its invoker to search for a handler associated with **bad_num**, while the **exit** statement transfers control to the end of the block to execute the handler associated with **done**.

Among all the exception handling mechanisms that have been proposed or implemented, CLU's is perhaps the simplest. Because it employs the single-level termination model, its semantics can be simulated in a programming language without an exception handling mechanism.

## 2.5. Ada's Exception Handling Mechanism

Ada's designers chose the termination model as the basis for its exception handling mechanism[Ichbiah 79]. Handlers are associated with exceptions at the end of a block, a subprogram body, a package body, or a task body. A handler at the end of a package body applies only to the initialization sequence of the package and not to subprograms in the package. When one of the declared exceptions is raised, the execution of the block (or the subprogram body, etc.) is abandoned. If there is a handler for that exception at the end of the block (or subprogram body, etc.), the handler is executed, finishing the execution of the whole block (or subprogram body, etc.). If no matching handler is found, the same exception is raised again at the point following the block (or in the calling subprogram). The propagation of the exception continues along the dynamic calling chain until either a matching handler is found or a task boundary is encountered.

Since unhandled exceptions propagate automatically along dynamic calling chain, it is possible for an exception to propagate outside its scope. As an example, consider the following package implementing a symbol table:

```
package symbol_table_manager is

    procedure enter_new_block;
    procedure leave_current_block;
    procedure store_symbol( ... );
    procedure lookup_symbol( ... );
    ...

end symbol_table_manager;

package body symbol_table_manager is
    ...
    package stack_pkg is
        ...
        stack_overflow : exception;

        procedure push( ... );  -- may raise stack_overflow
        ...
    end stack_pkg;

    package body stack_pkg is separate;

    procedure enter_new_block is
    begin
        push( ... );
    end enter_new_block;

    ...
end symbol_table_manager;
```

In the body of **symbol_table_manager**, there is an internal package **stack_pkg** which declares an exception **stack_overflow**. The procedure **enter_new_block** of **symbol_table_manager** invokes **push** defined in the stack package to push a new frame onto the stack. Since **push** may raise the exception **stack_overflow** which is not handled by **enter_new_block**, the exception is propagated outside its scope to a user program invoking **enter_new_block**. (The scope of **stack_overflow** starts from the exception's declaration and extends to the end of **symbol_table_manager**'s body). The automatic propagation of **stack_overflow** reveals that **symbol_table_manager** is implemented using stack. Thus we fail to achieve the design goal of hiding the implementation of the package from its users. Since Ada does not require that subprogram headings list exceptions that can be raised in their bodies, the behavior of a subprogram in Ada can only be understood by examining its implementation.

In Ada, handlers can be attached to blocks but not to statements. This often causes a program to be cluttered with blocks to insert handlers in the middle of statement lists. Ada's exceptions are not allowed to receive parameters; all communication between the signals and handlers must be accomplished through global variables.

### 2.6. Black's Thesis

Black[Black 83] argued that exception handling is neither necessary nor desirable. It is unnecessary because one can always use procedure parameters to replace a resumption–type handler, and use multiple result types along with explicit testing in an invoker to replace the termination of a signaler. He wrote in his concluding chapter:

> *"The fact remains that exception handling mechanisms have been proposed and implemented, and we may therefore ask what facility they add to a programming language. The answer is that they are a new control structure, in some languages carefully restricted in application, and in others so general as to replace the goto."*

In Black's proposal, functions return result values with union (i.e., **oneof**) types. Exceptions are declared as enumerated types, each containing a single value, e.g.,

```
type underflow is new singleton;
type overflow  is new singleton;

function stack_top_elem( S : int_stack )
                        return oneof( integer, underflow);
```

Boolean functions determine the type of the current value of a variable, e.g.,

```
function is_integer( v: oneof(integer, underflow) ) return boolean;
function is_underflow( v: oneof(integer, underflow) ) return boolean;
```

Thus, instead of using termination-type exception handlers, an invoker tests the result of a function call in the following fashion:

```
        I : integer;
        element : oneof( integer, underflow );

    begin
        element := stack_top_elem( S );
        if is_underflow(element) then
            error_message( "Stack underflow occurred" );
        else
            I := to_integer( element );
        end if;
```

where `to_integer` is a function that converts a union-type variable to a variable of type integer.

To replace resumption-type exception handlers, Black proposed using parametric procedures. An invoker of a procedure needs to supply handler procedures as actual parameters to the invoked procedure. These handler procedures can then be invoked to handle the exceptions that may be raised in the called procedure.

It is worth pointing out that Black's method does not replace all exception handling mechanisms. Indeed, he intentionally avoided applying his proposed method to a mechanism like Goodenough's or Yemini's[Yemini 85] where a handler has the freedom to choose whether to terminate or resume the signaler. His method prevents a signaler from deciding whether to terminate itself and return a special value representing the exception being raised, or to invoke a procedure parameter and handle the exception.

## 2.7. Dony's Mechanism

Dony proposed an exception handling mechanism for object-oriented programming languages[Dony 89]. In Dony's mechanism, exceptions are *classes* (i.e., types) which are instances of a dedicated *meta-class* (i.e., "generic type"). Exceptional events are *objects* instantiated from exception classes. An exception class can have its own *slots* (i.e., fields in a type) and *methods* (i.e., operations defined on a type). An exception object can be inspected, modified, or enriched as other *first class objects* in an object-oriented language. Exception classes are organized in a hierarchy, and each instance of a class *inherits* the properties of its *ancestors*. An instance can also *overload* certain properties of its ancestors.

Figure 1 shows an example of how exception classes are organized in a hierarchy. The nodes in the graph denote exception classes, and direct edges denote relationships between classes and subclasses. The exception class **exceptional_event** is the ancestor of all other exception classes shown in the graph. It is created by instantiating the exception meta-class **exception_class** (not shown in Figure 1). **exceptional_event** has two subclasses: **fatal_event** and **proceedable_event**. A **fatal_event** exception has a method for termination, while **proceedable_event** has one for resumption. The exception class **error** is the set of exceptional events for which resumption is not allowed, whereas exception class **warning** is the set of exceptional events for which resumption is mandatory. The exception class **exception**, as a subclass of both **fatal_event** and **proceedable_event**, is the set of the exceptional events that allows both termination and resumption. This is made possible by multiple inheritance. In Figure 1, the exception **window_large_than_screen** is a user-defined exception that refines the system

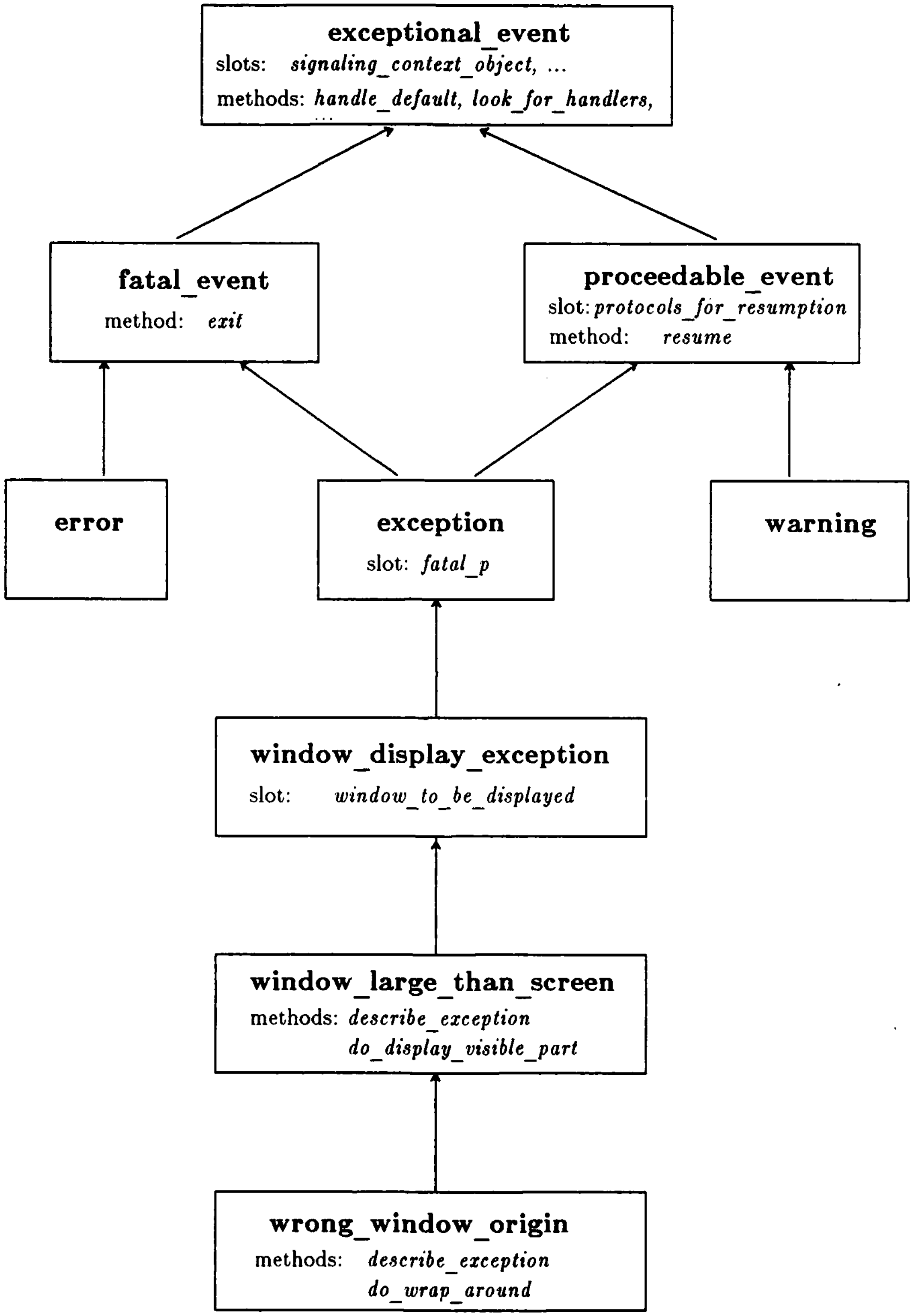



**Figure 1. An Exception Hierarchy**

defined exception `window_display_exception`.

Note that in the definition of exception class **wrong_window_origin**, the method `describe_exception` defined in its ancestor **window_large_than_screen** is overloaded. If a message is sent to the method **describe_exception** of a **wrong_window_origin** object, the method defined in **wrong_window_origin** is invoked.

Signaling an exception during execution of a method (of any object in the system) is accomplished by instantiating the corresponding exception class to create an exception object. After the exception object is created, a *message* is sent to one of its methods (e.g., **look_for_handlers**) asking it to locate and execute an appropriate handler. When an exception object is created, some of its slots can be explicitly assigned, while others take default values. In the following code,

```
if [ [w length] > [oself length] ] then
   [window_large_than_screen signal window_to_be_displayed : w]
```

**signal** instantiates the exception class **window_large_than_screen** to create an exception object and assigns the object's **window_to_be_displayed** slot the value `w`. **signal** then sends the instance the message **look_for_handlers** understood by all instances of exceptions to invoke a handler.

Since exceptions are organized hierarchically, invoking a **signal** primitive may signal multiple exceptions if the exception being signaled has subclass exceptions. Similarly, since all handlers are aware of the exception hierarchy, defining a handler for an exception amounts to defining a handler for all exceptions that are subclasses of it.

### 2.8. Summary

In the preceding sections, we surveyed several influential exception handling mechanisms proposed or implemented in the past. Space considerations preclude our covering other relevant mechanisms in this brief survey, e.g., Yemini's replacement model[Yemini 85] in which a handler uses its result to replace the result of expression raising the exception; Knudsen's proposal[Knudsen 87] for using sequels (essentially procedures that are passed as parameters) to handle exceptions raised in nested blocks; and Mesa's exception handling mechanism[Mitchell 79].

All implemented exception handling mechanisms associate handlers with exceptions via control components in a program, either dynamically by executing a statement making a handler available for a particular exception or statically by attaching a handler to an executable program unit. Attaching handlers to control statements clutters source text in much the same way that testing parameters for suitability as inputs and significance as results does. Generally, these mechanisms fail to achieve the goal of segregating normal algorithmic processing from error processing in a program.

Without a clear definition of exceptions, programs treat control-oriented exception handling mechanisms as just another control structure. Although exceptions can be signaled in response to software failures, they can also be used in normal processing situations. Thus in implementing algorithms, programmers are forced to choose from a wider set of primitive language features. Exception processing is fault-prone because it is the least well documented and tested part of an interface[Horning 79].

Nonetheless, we remain sanguine about the usefulness of exception handling mechanisms in identifying and segregating code that is executed in special circumstances. By equating exceptions with system insufficiencies as Black does and associating handlers with objects rather than control structures as Levin and Dony propose, exception handling mechanisms may still prove to be valuable implementation aids.

# CHAPTER 3

# Problems with Existing Mechanisms

This chapter discusses some general problems of control–oriented exception handling mechanisms and reports specific results of a case study of the use of exception handling in Ada programs.

## 3.1. Multiple Exit Points from Compound Statements

In a structured programming language, control–flow structures observe the "one–in, one–out" rule for regulating control flow (i.e., each construct has a single entry point and a single exit point). If **return** statements are forbidden in subprograms, then functions and procedures also observe this rule. However, exceptions add a large number of potential exit points. Beside the original exit point, every place where an exception can be raised may also transfer control out of the structure. A loop in Ada

```
sum := 0;
while not end_of_file loop
    get( n );
    sum := sum + n;
end loop;
put( sum );
```

contains at least three abnormal exit points in addition to its normal exit point: `get(n)` could raise `data_error`, the addition operator could raise `numeric_error`, and the assignment operator could raise `constraint_error`.

### 3.2. Inter–Module Coupling

Control–oriented exception handling mechanisms also increase the strength of inter–module coupling by adding exceptions to the interfaces of modules. Often the exceptions are implicit interface components. Although handlers are generally associated with control units statically, exceptions raised in procedure bodies are not listed in procedure headings. In languages with automatic propagation of exceptions, it may not be possible to tell if exceptions are part of an interface even after examining the procedure's body.

### 3.3. Mixed Algorithmic and Exceptional Code

Control–oriented exception handling mechanisms are designed to separate the code dealing with "normal" computation from the code dealing with "exceptional cases." Under different mechanisms, a handler can be attached to the end of an expression, a statement, a block, or a subprogram. No matter what scheme is used, handlers are still embedded in statement lists. Ada places exception handling code at the end of blocks. When a new handler is needed for an exception, blocks must be nested inside one another. The following example

```
get( n );
begin
    factor := size * n;
    sum := sum + factor;
exception
    when constraint_error => handler_1;
end;
```

shows a block that contains two statements performing arithmetic operations. If **constraint_error** is raised by either of the assignment operators, **handler_1** is executed. However, if different handling actions are needed for `numeric_errors` raised by different arithmetic operators, additional blocks must be introduced:

```
get( n );
begin
    begin
        factor := size * n;
    exception
        when numeric_error => handler_2;
    end;
    begin
        sum := sum + factor;
    exception
        when numeric_error => handler_3;
    end;
exception
    when constraint_error => handler_1;
end;
```

The nested blocks added for the sole purpose of attaching handlers interleave algorithmic and exception handling code.

Replacing a statement with a block having an attached handler may also lead to surprising results. In the example in Section 3.1, the summation of numbers read from the input file will never be printed because encountering end of file raises the exception `end_error` terminating the block enclosing the loop without executing the `put` statement. Correcting this problem by introducing a block with a handler for `end_error` causes different problems.

```
sum := 0;
loop
    begin
        get( n );
    exception
        when end_error => put( sum );
    end;
    sum := sum + n;
end loop;
```

This solution results in an unbounded loop because after handling the exception **end_error**, only the block, not the loop, is terminated.

### 3.4. Difficulties for Code Optimization

Control-oriented exception handling complicates code optimization; improvements based on code motion are inhibited if the effects of exception handling need to be guaranteed. For example, when using strength reduction to optimize a simple loop, some computations will be introduced prior to the loop's entry point. If these computations raise exceptions, the behavior of the loop could be changed.

Consider the following code segment:

```
begin
    I := C;
    while I <= 10 loop
        N := I * A;
        I := I + B;
    end loop;
exception
    when Numeric_Error =>
                ...
end;
```

After applying strength reduction, this code is transformed into:

```
begin
    I := C;
    t1 := I * A;
    t2 := A * B;
    while I <= 10 loop
        N := t1;
        I := I + B;
        t1 := t1 + t2;
    end loop;
exception
    when Numeric_Error =>
                ...
end;
```

By substituting cheaper addition operations for more expensive multiplication operations, the loop executes faster. This transformation poses no problem as long as no exceptions are raised in either version. However, suppose `A*B` produces an overflow and the predefined exception `numeric_error` is raised. In the case where the loop is not entered, the original loop will not raise an exception, but the optimized loop will. Thus, in the presence of exceptions, conventional optimization methods cannot guarantee preservation of the semantics of an Ada program.

As another example, consider how exceptions in Ada affect the semantics of `out` and `in out` parameters of subprograms. If a subprogram is terminated by an exception, then the values of `out` and `in out` parameters are not guaranteed. While scalar parameters will not have been updated, aggregate parameters may or may not be changed since a compiler may adopt either reference or copy-in, copy-out implementation strategies.

### 3.5. An Analysis of Exception Handling in Ada Programs

To investigate the use of Ada's exception handling features in application programs and assess their impact on programming practices, we analyzed about two dozen programs in the Simtel20 Ada Repository. The exception handlers found in these programs were divided into the following seven categories:

1). Sending error messages (including error logging, etc.);

2). Propagating exceptions;

3). Initializing and/or finalizing operations on objects (e.g., open/close a file,

**Usage of Exception Handling Mechanism in Ada Programs**

| File Name | Stmt Number | Handler Number | Stmt in Handlers | Error Message | Propa-gation | Init & Final | Term & Abort | Change Globals | Ignore (null) | Control & Others |
|---|---|---|---|---|---|---|---|---|---|---|
| expert.ada | 400 | 0 | | | | | | | | |
| mins.src | 856 | 20 | 45 | 6 | 7 | 22 | 3 | 2 | 2 | 3 |
| ed2.src | 771 | 11 | 42 | 20 | 4 | 13 | 2 | 2 | | 1 |
| wpcommon.src | 1552 | 9 | 9 | | | | 5 | 2 | 2 | |
| wpcrt.src | 2197 | 0 | | | | | | | | |
| wpformat.src | 4902 | 31 | 51 | 19 | 2 | 7 | 17 | 6 | | |
| wpeditor.src | 2035 | 39 | 59 | 18 | 3 | 20 | | 9 | 3 | 6 |
| ftp.src | 2993 | 179 | 334 | 152 | 153 | 14 | 4 | 7 | 4 | |
| iface.src | 573 | 9 | 10 | 5 | 5 | | | | | |
| smtp.src | 1001 | 71 | 118 | 66 | 46 | 5 | | | | 1 |
| smtpwicat.src | 1139 | 70 | 118 | 68 | 47 | 3 | | | | |
| tcpstand.src | 217 | 1 | 1 | 1 | | | | | | |
| tcpsub.src | 3981 | 141 | 224 | 218 | 6 | | | | | |
| tcptest.src | 667 | 5 | 17 | 17 | | | | | | |
| tcpwicat.src | 2706 | 168 | 252 | 233 | 6 | | 12 | | 1 | |
| telnet.src | 2476 | 94 | 198 | 104 | 94 | | | | | |
| telwicat.src | 2242 | 95 | 188 | 97 | 91 | | | | | |
| wicatmisc.src | 297 | 3 | 3 | 2 | | | | 1 | | |
| form2.src | 2869 | 160 | 320 | 66 | 90 | 12 | 7 | 21 | 32 | 92 |
| formtest.src | 163 | 2 | 2 | 2 | | | | | | |
| compord.src | 1052 | 44 | 140 | 52 | 2 | 1 | 5 | 2 | 12 | 66 |
| mman.src | 784 | 27 | 31 | 8 | | 7 | 4 | 8 | 4 | |
| mmgr.src | 2633 | 209 | 398 | 158 | 181 | 33 | 1 | 9 | 3 | 13 |
| manpower.src | 350 | 13 | 17 | 11 | | 6 | | | | |
| pplanner.src | 5112 | 161 | 480 | 266 | 89 | 35 | 45 | 3 | 7 | 35 |
| tracker.src | 4343 | 97 | 325 | 294 | 10 | 6 | | 14 | 1 | |

**Table 1. Usage of Exception Handling Mechanism in Ada Programs**

set/flush a buffer, etc.);

4). Terminating a block or sub-program or aborting a task;

5). Changing the values of global variables to record the significance of raised exceptions for later treatment;

6). Ignoring the exception being raised (null handlers); or

7). Performing significant repair or diagnostic actions (e.g., determining the site where a **constraint_error** is raised).

Tables 1 and 2 summarize our results. Most of the handler actions apply only the simplest form of exception handling, such as propagating the exception until termination, or printing error messages without doing anything else. Only two programs have more than 11% of their exception handling statements belonging to the last category (i.e., non-trivial algorithms).

Many of these simple handler actions can be simulated with other features of Ada. Error messages can be printed where the errors are detected by substituting handler bodies for **raise** statements. Propagating exceptions can be simulated by returning special values. Terminating a subprogram can be accomplished with a **return** statement. A null handler is a strong indication that the piece of code can be simply rewritten without exception handling. Changing the values of global variables to record the significance of raised exceptions shows a severe defect in Ada's exception handling mechanism. The global variables introduced tend to increase the complexity of a program by causing the modules in the program to be strongly coupled.

It is interesting to note that some programs never use any exception handling at all. This indicates either the authors were not comfortable with Ada's exception handling

| Distribution of Exception Handling Actions (in percentage) | | | | | | | |
|---|---|---|---|---|---|---|---|
| File Name | Error Message | Propa– gation | Init & Final | Term & Abort | Change Globals | Ignore (null) | Control & Others |
| expert.ada | | | | | | | |
| mins.src | 13.3 | 15.6 | 48.9 | 6.7 | 4.4 | 4.4 | 6.7 |
| ed2.src | 47.6 | 9.5 | 31.0 | 4.8 | 4.8 | | 2.4 |
| wpcommon.src | | | | 55.6 | 22.2 | 22.2 | |
| wpcrt.src | | | | | | | |
| wpformat.src | 37.3 | 3.9 | 13.7 | 33.3 | 11.8 | | |
| wpeditor.src | 30.5 | 5.1 | 33.9 | | 15.3 | 5.1 | 10.2 |
| ftp.src | 45.5 | 45.8 | 4.2 | 1.2 | 2.1 | 1.2 | |
| iface.src | 50.0 | 50.0 | | | | | |
| smtp.src | 55.9 | 39.0 | 4.2 | | | | 0.8 |
| smtpwicat.src | 57.6 | 39.8 | 2.5 | | | | |
| tcpstand.src | 100.0 | | | | | | |
| tcpsub.src | 97.3 | 2.7 | | | | | |
| tcptest.src | 100.0 | | | | | | |
| tcpwicat.src | 92.5 | 2.4 | | 4.8 | | 0.4 | |
| telnet.src | 52.5 | 47.5 | | | | | |
| telwicat.src | 51.6 | 48.4 | | | | | |
| wicatmisc.src | 66.7 | | | | 33.3 | | |
| form2.src | 20.6 | 28.1 | 3.8 | 2.2 | 6.6 | 10.0 | 28.7 |
| formtest.src | 100.0 | | | | | | |
| compord.src | 37.1 | 1.4 | 0.7 | 3.6 | 1.4 | 8.6 | 47.1 |
| mman.src | 25.8 | | 22.6 | 12.9 | 25.8 | 12.9 | |
| mmgr.src | 39.7 | 45.5 | 8.3 | 0.3 | 2.3 | 0.8 | 3.3 |
| manpower.src | 64.7 | | 35.3 | | | | |
| pplanner.src | 55.4 | 18.5 | 7.3 | 9.4 | 0.6 | 1.5 | 7.3 |
| tracker.src | 90.5 | 3.1 | 1.8 | | 4.3 | 0.3 | |

**Table 2. Distribution of Exception Handling Actions**

mechanism or they felt the programs could be better constructed without exception handling. There are only a few programs with deeply nested exception handlers, and not surprisingly, these programs are very hard to understand.

For our case study, we examined **compord.src**, one of the two programs with non-trivial handlers. This program calculates the correct compilation order of Ada source program units. One of its procedures, put_info_in_dag, uses the data type direct acyclic graph, to represent objects containing compilation dependencies of Ada program units. A dag object named withs_dag is used to record compilation dependencies derived from the with clauses preceding Ada compilation units. If a new edge is added to withs_dag, a cycle occurs. The newly added nodes and edges are entered into cycle_dag for error reporting later. Three exceptions are declared in dag_pkg: illegal_node is raised when a node is not in a dag, or when it is and should not be; duplicate_edge is raised by attempts to add an edge already in the graph; and makes_cycle is raised if a newly added edge would cause a cycle. A slightly edited version of the body of this procedure appears below. Two procedures, add_node_to_dag and add_to_cycle_dag, have replaced in-line code to reduce the length of the code and to make future comparisons between exception handling methods more fair.

```
1  begin
2     label := ... ;
3     value := wdag.get_value( withs_dag, parent_node );
4     ...
5     if not gen_inst then ...
6        wdag.set_value( withs_dag, parent_node );
7     end if;
8  exception
9     when wdag.illegal_node => ... ; add_node_to_dag( withs_dag, ... );
10 end;
11 ... ;
12 while id_list_pkg.more(i) loop
13    id_list_pkg.next( i, with_name );
14    begin
15       ...
16       wdag.add_node( withs_dag, with_node );
17       wdag.add_edge( withs_dag, parent_node, with_node );
18    exception
19       when wdag.illegal_node =>
20          begin
21             wdag.add_edge( withs_dag, parent_node, with_node );
22          exception
23             when wdag.makes_cycle =>
24                begin
25                   add_to_cycle_dag(cycle_dag, parent_node, with_node);
28                exception
29                   when idag.illegal_node | idag.makes_cycle => null;
30                end;
31             when wdag.duplicate_edge => null;
32          end;
33       when wdag.makes_cycle =>
34          begin
35             add_to_cycle_dag( cycle_dag, parent_node, with_node );
38          exception
39             when idag.illegal_node | idag.makes_cycle => null;
40          end;
41    end;
42 end loop;
```

This program has four blocks containing exception handlers; three of these blocks are nested within one another. Handler responses to exceptions vary for different data objects. For example, when **makes_cycle** is raised by **add_edge** on line 17, the signaler manipulates **withs_dag**. The handler (on line 33) puts related information into cycle_dag. A similar situation exists when makes_cycle is raised on line 21 and handled on line 23. However, when makes_cycle is raised inside add_to_cycle_dag on line 25 and 35, the signalers are processing cycle_dag. In such cases, the handlers (on lines 29 and 39) ignore the exception.

Examining the code more carefully, we found the exception handling code on lines 33–40 to be unreachable. In order for control to arrive at line 33, the procedure add_edge (line 17) must raise the exception makes_cycle. This implies that procedure add_node (line 16) completed without raising any exceptions that would have terminated the block on lines 14–41. Thus, add_node's argument, with_node, is a fresh node just added into withs_dag. In addition, with_node must be a node different from parent_node or illegal_node would have been raised. Since there is no edge connected to with_node when it is added to withs_dag and with_node is different from parent_node, adding an edge from parent_node to with_node will never generate a cycle.

This analysis indicates that Ada's exception handling mechanism is not being used very effectively. Most handlers have actions that could easily be simulated by other features of Ada. The program chosen for our case study has such complex exception handling that unreachable handlers went unnoticed.

# CHAPTER 4

# The New Mechanism

In the previous chapters, we showed that control–oriented exception handling mechanisms lack clear guidelines of use, duplicate existing language capabilities, interact with existing language features in undesirable ways, and seem to be underutilized by programmers. In this chapter, we propose a new mechanism in which exceptions and handlers are associated with data types and objects rather than control features.

## 4.1. A Data–Oriented Exception Handling Mechanism

We define *exceptions* as events arising during the execution of an operation where more system resources are required to represent the result. We associate exceptions with types rather than control structures. Generally, users of operations know best how to respond to exceptions raised[Parnas 76]. In the examples we have examined (e.g., **compord.src** in the previous chapter), different responses are required for different objects. Thus handlers should be associated with objects in declarations. Control–oriented exception handling mechanisms introduce extra testing code to distinguish data objects and multiple handlers to cope with the same exception raised for different objects. In the procedure **put_info_in_dag** shown at the end of Chapter 3, responses to exceptions vary for different data objects. When **makes_cycle** is raised, if the signaler is processing **withs_dag** (e.g., line 17), the handler invokes **add_to_cycle_dag** (e.g., the block on lines 34–40); if the signaler is processing **cycle_dag** (e.g., line 35), the handler ignores the exception (e.g., line 40). In addition, handling code on lines 23–30 is an exact copy of that on lines 33–40 due to duplicated call to **add_edge** (line 21) inside the handler for **illegal_node**.

Our design is based on the programming language Ada[Ada 82], particularly its **package** construct which is useful in implementing user-defined data types. The following package specification defines the data type **stack**:

```
generic
    type elem_type is private;
    tentative_size_limit : positive;    -- tentative initial size
package stack_pkg is

    type stack is limited private;

    procedure create( S : out stack );
    procedure push( S : in out stack; E : elem_type );
    procedure pop(  S : in out stack; E : out elem_type );
    procedure copy( S : stack; T : out stack );
    function  is_empty( S : stack ) return boolean;
    function  equal( S, T : stack ) return boolean;
    function  size( S : stack )     return natural;
    function  max_size( S : stack ) return natural;

    procedure expand( S : in out stack; amount : positive );

private

    type stack_object;
    type stack is access stack_object;

end stack_pkg;
```

An abstract stack object has unlimited size, although an initial size is specified. The procedure **expand** can be called to expand the pre-allocated storage for a stack S by **amount** percent.

Our *data-oriented exception handling mechanism* is built into Ada by extending the definition of the base language. We introduce two clauses, **#exception** and **#when**, to declare exceptions and associate handlers with exceptions for objects; as well as an additional statement, **#raise**, to signal exceptions.

### 4.1.1. Declaring Exceptions

Exceptions are defined on data types by attaching an **#exception** clause to the type definition exported from the specification part of a package. The syntax of the **#exception** clause is given by modifying the following production in Ada Language Reference Manual (LRM) Section 7.4:

```
private_type_declaration ::=
        type identifier [ discriminant_part ] is [ limited ] private;
```

to

```
private_type_declaration ::=
        type identifier [ discriminant_part ] is [ limited ] private
                                             [ exception_clause ] ;

exception_clause ::= #exception exception_formal_specification_list

exception_formal_specification_list ::=
        exception_formal_specification { , exception_formal_specification }

exception_formal_specification ::= identifier formal_part
```

**Table 3. Syntactic Specification of #exception Clauses**

Note that the syntactic categories *discriminant_part*, *identifier*, and *formal_part* are defined as in LRM, Section 3.7.1, 2.3, and 6.1, respectively. As an example, the following **#exception** clause can be attached to the data type **stack** declared in the package **stack_pkg**:

```
type stack is limited private
      #exception overflow( S : in out stack; place : string ),
                 storage_exhausted( S : in out stack );
```

Two exceptions **overflow** and **storage_exhausted** define system insufficiencies involving **stack** objects. These exceptions are declared by attaching an **#exception** clause to the type declaration, emphasizing that only the operations defined on the type can raise the exceptions. Ada exceptions declared in the visible part of a package can be raised not only by any subprogram defined in the package, but also by any other subprogram using the package. An operation on an object of type **stack** will raise the exception **overflow** when the object grows beyond its size limit. The exception can be handled by increasing the size of the stack and allocating more storage for the object. Once **overflow** has been handled, the original computation can be resumed. In the event that all system memory resources have been exhausted, the more severe exception **storage_exhausted** is raised.

All exceptions take parameters to facilitate communication between their signalers and handlers. The two exceptions declared in the previous example both take a parameter of type **stack** to indicate which stack object needs more storage. **overflow** takes an additional parameter **place** of type **string** to identify the signaler. The first parameter in the formal parameter list of an exception declaration *must* belong to the type currently being declared. This parameter is used to denote the data object for which the exception is raised.

### 4.1.2. Raising Exceptions

In the body of a package implementing a user-defined data type, exceptions can be raised within the statement sequence of an operation. To raise an exception, a **#raise** statement is executed for a particular data object.

To give the syntax of **#raise** statements, we need to modify the following production in LRM, Section 5.1:

```
simple_statement ::= null_statement
        | assignment_statement  | procedure_call_statement
        | exit_statement        | return_statement
        | goto_statement        | entry_call_statement
        | delay_statement       | abort_statement
        | raise_statement       | code_statement
```

to

```
simple_statement ::= null_statement
        | assignment_statement  | procedure_call_statement
        | exit_statement        | return_statement
        | goto_statement        | entry_call_statement
        | delay_statement       | abort_statement
        | raise_statement       | code_statement
        | data_oriented_raise_statement

data_oriented_raise_statement ::=
        #raise identifier actual_parameter_part
```

**Table 4. Syntactic Specification of #raise Statements**

The syntactic category *actual_parameter_part* is defined in Ada Language Reference Manual, Section 6.4. The numbers, types, and positions of the actual parameters supplied in a **#raise** statement should agree with those of the formal parameters declared in the corresponding **#exception** clause.

Unlike in Ada where a **raise** statement can be executed wherever a statement can be invoked, our **#raise** statement can only occur in the body of a package implementing a user-defined data type with which the exception is declared. For example, in package **stack_pkg**, the operation **push** can raise an exception **overflow** when the pre-allocated storage for a stack object does not have enough room to accommodate more items.

```
procedure push( S : in out stack; E : elem_type ) is
begin
    if { there is not enough room to hold the new item } then
        #raise overflow( S, "in procedure PUSH" );
    end if;
    ...                           -- add item E to the top of S
end push;
```

Note that an exception is usually raised for an object passed to an operation as a parameter. In the above example, the exception `overflow` is raised on `S` which is a parameter of the operation `push`. Users of the `stack_pkg` are able to declare their own stack objects and associate handlers for `overflow` with these objects. Exceptions can also be raised for local objects inside the body of a package.

### 4.1.3. Binding Handlers to Objects

Handlers are associated with exceptions in the declaration of a data object. A `#when` clause can be attached to the declaration of an object, supplying one or more handlers to the corresponding exceptions defined on the data object. A handler body is limited to a single statement.

The syntax for object declarations in Ada found in LRM, Section 3.2 is as follows:

object_declaration ::=
  identifier_list : [ **constant** ] subtype_indication [ := expression ] ;
| identifier_list : [ **constant** ] constrained_array_definition [ := expression ] ;

To give the syntax for the `#when` clauses, we need to modify the above production to:

```
object_declaration ::=
        identifier_list : [ constant ] subtype_indication [ := expression ]
                                                    [ when_clause ] ;
      | identifier_list : [ constant ] constrained_array_definition
                                                    [ := expression ] ;

when_clause ::= #when handler_association_list

handler_association_list ::= handler_association { , handler_association }

handler_association ::=
        exception_formal_specification => handler_statement
```

**Table 5. Syntactic Specification of #when Clauses**

The definition for the syntactic category *exception_formal_specification* can be found in Table 3 of this chapter; while *actual_parameter_part* is defined in LRM, Section 6.4.

When associating a handler with an exception, the exception name and its formal parameter list are specified to the left of the symbol "=>". In the exception specification, the types and positions of the formal parameters must be exactly the same as those appearing in the corresponding **#exception** clause. However, any names can be used for the formal parameters. In a handler, the name of a formal parameter of an exception appearing on the left of the "=>" symbol can be used as an actual parameter of a procedure call on the right of the "=>" symbol.

When an exception is raised for an object, if there is a handler associated with the exception, the handler will be executed and control then returns to the point following the **#raise** statement; however, if there is no handler associated with the exception, the execution of the whole program is terminated.

As an example, the following code segment declares two integer stacks S1 and S2, and associates a handler with the exceptions **overflow** and **storage_exhausted**:

```
with stack_pkg;
procedure main is

    package integer_stack is new stack_pkg( integer, 20 );
    use integer_stack;

    procedure urgent_action;

    S1, S2 : stack
               #when overflow( S : in out stack; place : string )
                                                  => expand( S, 40 ),
                     storage_exhausted( T : in out stack )
                                                  => urgent_action;
    procedure urgent_action is separate;
begin
    ...
end main;
```

When the exception **overflow** is raised by some operations (e.g., **push**) trying to update **S1** or **S2**, the operation **expand** is executed. If the execution of the handler **expand** succeeded, the storage for the object involved is expanded by 40%. Note that the actual parameter **S** of the procedure **expand** can denote either **S1** or **S2**, depending on which object is passed by the corresponding **#raise** statement. If **storage_exhausted** is raised, **urgent_action** is invoked.

If a handler is a visible operation of the data type, it can raise another exception during its execution. For example, **expand** can raise **storage_exhausted** if it realizes that there is no more system storage available to expand the stack. To avoid endless recursion, the handler is not permitted to raise the same exception with which it is associated. In the example given above, **expand** is not allowed to raise the exception **overflow**. Care must also be taken to prevent indirect recursion involving two or more handlers. A possible solution for this problem is to assign different degrees of severity to exceptions and force handlers to raise only more severe exceptions.

In the previous example, the exception **storage_exhausted** is associated with a handler **urgent_action** to prevent the program from being terminated should this exception be raised. Two possible alternative solutions that **urgent_action** can choose are:

- Abort the program after performing cleaning-up actions. Such actions include finalizing some data structures (e.g., closing open files), and issuing farewell messages;
- Initiate system garbage collection for the system-maintained heap.

In most cases, a handler for an exception is an exported operation declared in the visible part of a package implementing a user-defined data type. Since exceptions are defined for implementation insufficiencies occurring in operations of a data type and the representation of the type is hidden, operations defined in a package can handle exceptions most efficiently. However, a user may supply a procedure other than the operations exported by the data abstraction as the handler for an exception because he wants some special treatment for the exception. In the following example, the programmer discards the oldest values in a stack to make room for the newer items when **overflow** is raised:

```
with stack_pkg, text_io;    use text_io;
procedure main is

    package integer_stack is new stack_pkg( integer, 50 );
    use integer_stack;

    procedure makes_room( S : in out stack;
                          discard : positive; where : string  );
    S1, S2 : stack
           #when overflow( S : in out stack; place : string )
                                     => makes_room( S, 20, place );

    procedure makes_room ( S : in out stack;
                           discard : positive; where : string ) is
        part_to_keep : positive;
        T : stack;
        E : integer;
    begin
        put_line( "*** stack overflow occurred " & where );
        part_to_keep := integer( float(max_size(S)) *
                                   (1.0 - float(discard)/100.0) );
        create( T );
        for I in 1 .. part_to_keep loop    -- save upper part in T
            pop( S, E );
            push( T, E );
        end loop;
        while not is_empty(S) loop    -- throw away oldest elements
            pop( S, E );
        end loop;
        for I in 1 .. part_to_keep loop    -- move back to S
            pop( T, E );
            push( S, E );
        end loop;
        put_line( "*** bottom" & integer'image(discard)
                       & "%" & "of the stack discarded" );
    end makes_room;

begin
    . . .
end main;
```

Note that `makes_room` is not implemented efficiently because the user does not know the representation of the `stack` data type. However, this example shows that users do have flexibility in constructing their own handlers.

### 4.2. Compord.src Revisited

To demonstrate the impact our exception handling mechanism might have on programs, we rewrote **put_info_in_dag** shown in Chapter 3. Although the exceptions are not limited to system insufficiencies, marked improvement can still be observed in program structure. The transformed program declared exceptions on the **dag** data type as follows:

```
package dag_pkg is

    type dag is private
            #exception duplicate_node( g : dag; l : label ),
                       node_not_in_dag( g : dag; l : label ),
                       duplicate_edge( g : dag; l1, l2 : label ),
                       makes_cycle( g : dag; l1, l2 : label );
    ...
end dag_pkg;
```

All handler actions associated with exceptions raised concerning **cycle_dag** are null actions. Therefore, in the transformed version, we attach **null** handlers for **duplicate_node** and **makes_cycle** to **cycle_dag** in the object's declaration:

```
cycle_dag : idag.dag
            #when duplicate_node( g : dag; l : label ) => null,
                  makes_cycle( g : dag; l1, l2 : label ) => null;
```

The situation is not as simple for **withs_dag** since it is an in-out parameter with different exception bindings in other scopes. However, introducing a new local variable, **temp_dag**, and attaching handlers to it is straightforward:

```
temp_dag : wdag.dag
           #when duplicate_node( g : dag; l : label ) => null,
                 node_not_in_dag( g : dag; l : label ) =>
                              add_node_to_dag( g, l ),
                 duplicate_edge( g : dag; l1, l2 : label ) => null,
                 makes_cycle( g : dag; l1, l2 : label ) =>
                              add_to_cycle_dag( cycle_dag, l1, l2 );
```

The revised version of the procedure is shown below:

```
temp_dag := withs_dag;

label := ... ;
value := wdag.get_value( temp_dag, parent_node );
if not gen_inst then ...
    wdag.set_value( temp_dag, parent_node );
end if;
while id_list_pkg.more(i) loop
    id_list_pkg.next( i, with_name );
    ...
    wdag.add_node( temp_dag, with_node );
    wdag.add_edge( temp_dag, parent_node, with_node );
end loop;

withs_dag := temp_dag;
```

The original procedure and the revised version (including the extra procedures) have about the same number of statements. Since the two versions of the procedure accomplish the same task with the same algorithm, we should not expect this number to change greatly. However, the new version breaks the original code into three smaller procedures, resulting in better modularity and functionality. As for the complexity, the original version has up to three levels of nested handlers, one of which was unreachable. In contrast, the revised version has no handler code mixed with the main code of computation, thus emphasizing the main algorithm and enhancing readability. Sample execution on worse-case data show no difference in execution time between the two versions and approximately a 5% space penalty in the compiled code of the data-oriented version.

# CHAPTER 5

# Implementation

We have implemented a pre-processor for translating pseudo-Ada programs with data-oriented exception handling to logically equivalent Ada programs. Although there are some restrictions imposed by the features of the Ada language, the implementation of the pre-processor is still relatively straightforward because of the simplicity of our mechanism. More importantly, this pre-processor provides us with the necessary means for conducting experiments to investigate the effect of using different exception handling methods on program construction. In this chapter, we will discuss in detail the design decisions made and implementation methods adopted in implementing the pre-processor.

## 5.1. Some Implementation Issues

The semantics of our exception handling mechanism strongly suggests passing handler procedures as parameters when the objects to which the handlers are bound are actual parameters. For example, the following statement invoking the **copy** operation in **stack_pkg**:

```
copy( S1, S2 );
```

where `S1` and `S2` belong to type **stack** can be translated to:

```
copy( S1, over1, stor1, S2, over2, stor2 );
```

where $\texttt{over}_i$ and $\texttt{stor}_i$ are handler procedures supplied to the declaration of $\texttt{S}_i$:

```
S1 : stack
     #when overflow(S: in out stack; place: string) => over1( ... ),
           storage_exhausted(S: in out stack) => stor1( ... );

S2 : stack
     #when overflow(S: in out stack; place: string) => over2( ... ),
           storage_exhausted(S : in out stack) => stor2( ... );
```

**#raise** statements are translated to procedure calls on the appropriate procedure parameters. The pre-processor identifies the data object involved and uses the object name in conjunction with the exception name to locate the formal handler procedure supplied. Although this transformation scheme is intuitively appealing, it cannot be used since Ada does not support procedure parameters.

Another implementation technique associates natural numbers with data objects and exceptions, passes these numbers as additional parameters when an object is passed as an actual parameter, and invokes a dispatch procedure visible to both the caller and callee when the callee raises an exception. A dispatch procedure with two formal parameters identifying the object and exception would be used to invoke the appropriate handler.

```
procedure handler_dispatch( obj_numb : object_number;
                            exception_numb : exception_number ) is
begin
    case obj_numb is
        when 1 =>
            case exception_numb is
                when 1 => handler_1;
                when 2 => handler_2;
                ...
            end case;
        when 2 =>
        ...
    end case;
end handler_dispatch;
```

The pre-processor translates an invocation to **copy** into:

```
copy( S1, i, S2, j );
```

where i and j are object numbers assigned to **S1** and **S2** respectively. Assuming the exception number assigned to **storage_exhausted** is 2, the procedure **copy** in the body of **stack_pkg** is then translated to:

```
procedure copy( S : stack; obj1 : object_number;
                T : out stack; obj2 : object_number ) is
    ...
begin
    ...
    -- #raise storage_exhausted( T );
    handler_dispatch( obj2, 2 );
    ...
end copy;
```

This translation scheme has a severe drawback. For the scheme to work correctly, the dispatch procedure has to be *visible* to both a user subprogram and the operations (e.g., **copy**) in a package body. The dependence of the package body on the dispatch procedure results in the package only being used by one program for each compilation. A package is generally stored in a library, used by many programs, and is often implemented before a user program is developed. Although it is possible to make a dispatch procedure visible to both a user subprogram and a library package by putting it into a separate library package and be "**with**"ed by both the user subprogram and the library package, the library package must be recompiled every time it is used by some user subprogram.

A third translation scheme utilizes generic formal subprogram parameters to pass handler procedures from a user program to a library package implementing a data abstraction. With this scheme, a package specification is translated to:

```
generic
    ...
    with procedure formal_handler_1( ... );
    with procedure formal_handler_2( ... );
    ...
    with procedure formal_handler_n( ... );
package package_name is
    ...
end package_name;
```

When the generic package is instantiated, the pre-processor supplies actual procedures to match these generic formal procedures. The actual procedures are the collection of all handler procedures given in the **#when** clauses attached to object declarations. In the body of the package, a **#raise** statement is translated to an invocation of one of the generic formal procedures.

This translation scheme, like the other two discussed earlier, still fails to solve our problem. The total number of different handler procedures that could appear in a user subprogram cannot be predetermined at the time the generic library package is written. Therefore, it is impossible to determine the number of generic formal handler procedures before the package is used. Secondly, a **#raise** statement could be mapped to one of several generic formal handler procedures depending on which object is bound to the formal parameter appearing in the **#raise** statement. Finally, only the types visible to both the library package and a user subprogram can be used as formal parameters in a generic formal subprogram. Like the previous translation scheme, this approach couples library packages too tightly with user subprograms. The resulting library package cannot be shared among different user subprograms.

## 5.2. The Translation Scheme

After carefully examining the translation methods discussed in the previous section, we realized that the right approach must combine the strengths of the feasible methods. Our translation scheme is a hybrid of dispatch procedures and generic formal subprograms. We create dispatch procedures in a user subprogram and pass them to the package body as generic actual parameters. Each of the dispatch procedures deals with only one exception, checking the data object involved and selecting the right handler procedure for execution. Under this scheme, the number of generic formal subprograms declared in a package specification is exactly the same as the number of exceptions declared on the data type within the package specification. Inside the package body, a **#raise** statement is translated to an invocation of a corresponding generic formal procedure.

### 5.2.1. Numbering Objects

To distinguish different objects of a user-defined type so that a dispatch procedure associated with an exception is able to select a handler for execution, sequential positive numbers are assigned to different objects. When an object is passed as a parameter, the number is passed as well. If an exception is raised inside the body of an operation, the object number of the object involved in the exception is passed to a generic formal (dispatch) procedure dedicated to that specific exception. This dispatch procedure uses the object number to locate the corresponding handler for invocation.

### 5.2.2. Passing Addresses with Object Numbers

In a `#when` clause, a handler associated with an exception can have an actual parameter passed by a `#raise` statement. For example, the local variable `S4` in the following block is passed to `my_handler` when `overflow` is raised for it:

```
declare
    S4 : int_stack.stack
         #when overflow( S : in out stack; place : string ) =>
                                                my_handler( S, place ),
               storage_exhausted( S : in out stack ) =>
                     put_line( "storage exhausted, ignore it." );
begin
    . . .
end;
```

The pre-processor should translate `S` to `S4` before copying `my_handler` into a dispatch procedure. However, `S4` is a local object and not visible to the nonlocal dispatch procedure for overflow. To avoid this problem, we must pass addresses of objects from a user program to the operations in a package so that a `#raise` statement can pass addresses to a dispatch procedure. The original objects can then be reconstructed and used in a handler invocation. In a later section, we will discuss how the actual parameters in a handler are processed before the handler is put into a dispatch procedure.

Thus, whenever a subprogram is invoked with an object of a user-defined type as a parameter, the address of the object and the object number are added to the actual parameter list along with the object itself. To obtain the address of an object, the Ada attribute *address* can be used. As an example, the following is a call to the `copy` operation declared in `stack_pkg`:

```
copy( S, 4, S'address, T, 6, T'address );
```

where `S` and `T` are of type `stack` assigned object numbers `4` and `6`, respectively. The pre-processor makes sure that subprogram headings in a package specification, package body, or user program are translated correspondingly. For example, the procedure **copy** declared in the specification part of **stack_pkg** is translated to:

```
procedure copy( S: stack; zzz_obj1: integer; zzz_addr1: address;
                T: out stack; zzz_obj2: integer; zzz_addr2: address);
```

The variable names introduced by the pre-processor always start with the prefix zzz_.

### 5.2.3. Translating Package Specifications with #exception Clauses

When translating a package specification containing an **#exception** clause, the pre-processor uses the information provided in the **#exception** clause to construct a set of generic formal procedure parameters for the package. The text of the **#exception** clause attached to the exported type definition are placed in comments and a semicolon is appended to the end of the keyword **private** to terminate the type declaration. For instance, the following code segment can be found in **stack_pkg**:

```
    type stack is limited private;
--         #exception overflow( S : in out stack; place : string ),
--                    storage_exhausted( S : in out stack );
```

For each exception $\mathbf{expt}_i$ declared in the **#exception** clause, a generic formal procedure parameter named $\mathtt{zzz\_expt}_i$ is added before the package heading. Each formal parameter of the exception $\mathbf{expt}_i$ of the type being defined is translated to an integer parameter $\mathtt{zzz\_obj}_j$ and an **address** parameter $\mathtt{zzz\_addr}_j$. Other formal parameters of $\mathbf{expt}_i$ are simply copied over to the formal parameter list of $\mathtt{zzz\_expt}_i$ except their types are converted to type **address**. In order for an Ada compiler to recognize type **address**, the package **system** is made visible to the current package by introducing a

**with** clause before the first line of the package. For example, the heading of the package **stack_pkg** is changed to:

```
with system;    use system;
generic
    type elem_type is private;
    size : positive;

    with procedure zzz_overflow( zzz_obj1 : integer;
                                 zzz_addr1 : address; place: string);
    with procedure zzz_storage_exhausted( zzz_obj1 : integer;
                                          zzz_addr1 : address );
package stack_pkg is
    ...
end stack_pkg;
```

For each subprogram $\texttt{oper}_i$ exported from the package, if it has a formal parameter of type being defined, two additional formal parameters are added: integer $\texttt{zzz\_obj}_j$ and address $\texttt{zzz\_addr}_j$. The procedures specifications for **push** and **copy** in **stack_pkg** are translated to:

```
procedure push( S: in out stack; zzz_obj1: integer;
                                 zzz_addr1: address; E: elem_type );

procedure copy( S: stack; zzz_obj1: integer; zzz_addr1: address;
                T: out stack; zzz_obj2: integer; zzz_addr2: address);
```

### 5.2.4. Translating Package Bodies with #raise Statements

A package body containing **#raise** statements can only be translated by the pre-processor after the corresponding package specification has been processed. Each subprogram in the package body has its formal parameter list altered to match the changes made in the specification. The subprogram headings of both the visible operations exported from the package and the internal subprograms should be modified in this way.

Care must be taken to transform nested subprogram declarations. If additional parameters $\texttt{zzz\_obj}_j$ and $\texttt{zzz\_addr}_j$ are added to the heading of an inner subprogram, the names of these parameters must differ from those in the enclosing subprograms.

The pre–processor replaces a **#raise** statement for an exception **expr** with an invocation of the corresponding generic formal procedure **zzz_expr**. The actual parameters supplied to **zzz_expr** are obtained as follows: if an actual parameter $\texttt{ap}_i$ supplied to a **#raise** statement is of the type being defined, it is converted to the corresponding $\texttt{zzz\_obj}_i$ and $\texttt{zzz\_addr}_i$ as the actual parameters for **zzz_expr**; otherwise, it is translated to an address in the form of $\texttt{ap}_i$**'address** as the corresponding actual parameter for **zzz_expr**. The invocation of the generic formal procedure **zzz_expr** actually invokes a dispatch procedure in the user program. As an example, the following code segment shows how a **#raise** statement in the body of procedure **push** is translated by the pre–processor:

```
procedure push( S : in out stack; zzz_obj1 : integer;
                    zzz_addr1 : address; E : elem_type ) is
begin
    if S.top = S.storage'last then
 --      #raise overflow( S, "in procedure PUSH" );
        zzz_overflow( zzz_obj1, zzz_addr1, "in procedure PUSH" );
    end if;
    . . .
end push;
```

In a package body, if a subprogram is invoked with an actual parameter $\texttt{ap}_i$ of the type being defined, its additional actual parameters (i.e., an object number and an address) are added to the actual parameter list. The additional actual parameters are obtained by observing the following rule: if $\texttt{ap}_i$ is a formal parameter in an enclosing subprogram, the corresponding **zzz_obj**$_i$ and **zzz_addr**$_i$ are used; otherwise, the object number assigned to **ap**$_i$ as well as an address in the form of **ap**$_i$**'address** are supplied.

### 5.2.5. Translating User Programs with #when Clauses

A user program with **#when** clauses attached to declarations can only be processed after a package specification containing a corresponding **#exception** clause has been translated. The pre-processor scans the **with**-clauses preceding the program heading and uses the package names in the **with**-clauses to obtain information about exceptions declared by the package that the pre-processor stored in a file. The file contains the type name to which an **#exception** clause was attached and the names and parameter types of the exceptions declared for the type.

Translating a user program results in the construction of a set of dispatch procedures which are passed as generic actual subprogram parameters when a package is instantiated. The number of dispatch procedures required is exactly the number of exceptions declared in the corresponding **#exception** clause. The names of the dispatch procedures are constructed based on the names of exceptions declared in the **#exception** clause and the name of the package to be instantiated. The number and types of formal parameters required by a dispatch procedure is determined by the formal parameters of the corresponding exception **expt**. The formal parameters of **expt** that are of type for which the exception is defined are translated into two formal parameters: **zzz_obj**$_i$ of type **integer** and **zzz_addr**$_i$ of type **address** for the dispatch procedure. Other parameter names of **expt** are copied over as the parameter names of the dispatch procedure; however, their types are changed to **address**. The following code segment demonstrates the instantiation of the generic package **stack_pkg** to produce two stack types:

```
procedure zzz_int_stack_overflow( zzz_obj1 : integer;
                            zzz_addr1 : address; place : string );
procedure zzz_int_stack_storage_exhausted( zzz_obj1 : integer;
                                           zzz_addr1 : address );

package int_stack is new stack_pkg( integer, 20,
                                    zzz_int_stack_overflow,
                                    zzz_int_stack_storage_exhausted);

procedure zzz_real_stack_overflow( zzz_obj1 : integer;
                             zzz_addr1 : address; place : string );
procedure zzz_real_stack_storage_exhausted( zzz_obj1 : integer;
                                            zzz_addr1 : address );

package real_stack is new stack_pkg(real, 10 ,
                                    zzz_real_stack_overflow,
                                    zzz_real_stack_storage_exhausted);
```

The dispatch procedures collect all the handlers supplied to **#when** clauses appearing in a program (including local variables). If the source text is only scanned once, the dispatch procedures must be placed after the last line of the program. Ada's separate compilation facility easily resolves the problem of physical code placement by putting body stubs for dispatch procedures at the end of the main declaration list, and generating the code for them as separate compilation units after the program text.

The contents of a dispatch procedure **zzz_int_stack_overflow** for exception **overflow** declared in package **stack_pkg** in the following example is derived by collecting all **#when** clauses attached to the stack objects declared. The body of **zzz_int_stack_overflow** contains a **case** statement which selects a handler according to the object number passed to it.

```
with text_io, stack_pkg;     use text_io;
procedure main is
    zzz_abort : exception;
    ...
    procedure my_handler(S: in out int_stack.stack; zzz_obj1: integer;
                         zzz_addr1 : address; amount : positive );
    procedure last_wish;

    growth_rate : positive;
    S1, S2 : int_stack.stack;
--          #when overflow( S : in out stack; place : string )
--                             => my_handler( S, growth_rate ),
--                storage_exhausted( T : in out stack) => last_wish;
    ...
    procedure zzz_int_stack_overflow( zzz_obj1 : integer;
                  zz_addr1 : address; place : string ) is separate;
begin
    declare
        S3 : int_stack.stack;
--          #when overflow( S : in out stack; place : string )
--                                       => my_handler( S, 30 );
    begin
        ...
    end;
    ...
end main;

with unchecked_conversion;
separate( main )
procedure zzz_int_stack_overflow( zzz_obj1 : integer;
                    zzz_addr1 : address; place : string ) is
    type zzz_ptr is access int_stack.stack;
    function zzz_addr_to_ptr is new
                unchecked_conversion( address, zzz_ptr );
    zzz_p1 : zzz_ptr := zzz_addr_to_ptr( zzz_addr1 );
begin
    case zzz_obj1 is
        when  1 => my_handler( zzz_p1.all,  1,
                               zzz_addr1, growth_rate );
        when  2 => my_handler( zzz_p1.all,  2,
                               zzz_addr1, growth_rate );
        when  3 => my_handler( zzz_p1.all,  3, zzz_addr1, 30 );
        when others => raise zzz_abort;
    end case;
end zzz_int_stack_overflow;
```

Three stack objects `S1`, `S2`, and `S3` are declared with type `int_stack.stack` and assigned object numbers 1, 2 and 3, respectively.

An object of type **stack** can be declared without associating a handler for a specific exception. Accordin; to the semantics of our mechanism, if such an exception is raised on that object, the whole program should be terminated. This is realized by the "**when others**" arm of the **case** statement in the corresponding dispatch procedure which raises the Ada exception **zzz_abort**. Since there is no Ada exception handler for **zzz_abort**, termination of the program is guaranteed.

The parameters of a handler are translated before the handler is ready to be put into a dispatch procedure. An actual parameter of a handler can be a formal parameter in the exception specification appearing before the "=>", a literal, or a global variable in the user program. Local variables cannot be used as actual parameters of a handler because they are not visible to a dispatch procedure declared as a first-level subprogram in a user program.

When constructing a handler in a dispatch procedure from a corresponding handler in a **#when** clause, the actual parameters that are global variables or literals are simply copied. However, parameters of a **#when** clause that are formal parameters of a handler require more attention. The pre-processor inserts code to use the addresses of objects supplied by **#raise** statements to retrieve a copy of the object. **unchecked_conversion** initializes an access variable so that it references an object with proper structure. The access variable can then be de-referenced to produce the object. In the dispatch procedure **zzz_int_stack_overflow**, **zzz_addr1** is converted to an access value referencing an object of type **int_stack.stack** and is then used to initialize **zzz_p1**. When invoking **my_handler**, the stack object is obtained by de-referencing **zzz_p1** (i.e., zzz_p1.all). The object number and address of a stack object is also passed in the invocation of `my_handler` so that any exception raised on the stack object can be properly handled.

# CHAPTER 6

# Program Verification

In this chapter, we present proof rules for our exception handling constructs. The Floyd/Hoare[Floyd 67][Hoare 69] axiomatic approach is used for expressing proof rules and operational specifications are given for package operations [Wulf 76]. A comparison between these rules and those proposed for control-oriented exception handling mechanisms demonstrates the simplicity of our approach.

## 6.1. The Operational Approach to Correctness of Modules

The operational approach to specification gives a recipe for implementing an operation with types and operations from well-defined abstract domains. For example, **sequences** can be defined informally as in Table 6[Wulf 76].

Using **sequences**, the abstract input and output assertions of an abstract operation **pop** in the package **stack_pkg** can be given as follows:

```
procedure pop( S : in out stack; E : out elem_type );
    -- β_pre  :  S = S' ∧  S ≠ <>
    -- β_post :  S = leader(S')  ∧  E = last(S')
```

The behavior of an operation exported from a package is specified using a pair of assertions about values in the abstract domain: an input assertion $\beta_{pre}$ and an output assertion $\beta_{post}$. Correspondingly, in the package body, the implementation of the same operation is defined with a pair of assertions about values of the concrete variables used to represent abstract objects: an input assertion $\beta_{in}$ and an output assertion $\beta_{out}$.

| | |
|---|---|
| $<s_1, \ldots, s_k>$ | denotes the sequence of elements specified; in particular, "$< >$" denotes the empty sequence, "nullseq." |
| $s \sim x$ | is the sequence which results from concatenating element x at the end of sequence s. |
| length(s) | is the length of the sequence "s." |
| first(s) | is the first (leftmost) element of the sequence "s." |
| trailer(s) | is a sequence derived from "s" by deleting the first element. |
| last(s) | is the last (rightmost) element of the sequence "s." |
| leader(s) | is a sequence derived from "s" by deleting the last element. |
| seq(V,n,m) | where "V" is a vector and "n" and "m" are integers, is an abbreviation for the sequence "$<V_n, V_{n+1}, \ldots, V_m>$"; alternately, seq(V,n,m) = seq(V,n,m−1) $\sim V_m$. |
| Note: first, trailer, last, and leader are undefined for "$< >$" | |

**Table 6. Informal Definition of Sequences**

Suppose in the body of **stack_pkg** the type **stack** is implemented as a record with fields **storage** and **top**, where **storage** is a one-dimensional array and **top** is the index of the top element (if any) in the stack. If a stack is empty, its **top** component has value 0.

```
package body stack_pkg is

    type vector is array( positive range <> ) of elem_type;
    type vector_pointer is access vector;

    type stack_object is record
        storage : vector_pointer;
        top     : natural;
    end record;

    procedure push( S : in out stack; E : elem_type ) is
    begin
        if S.top = S.storage'last then
            #raise overflow( S, "in procedure PUSH" );
        end if;
        S.top := S.top + 1;
        S.storage(S.top) := E;
    end push;

    . . .

end stack_pkg;
```

A *representation function* mapping concrete values to abstract values relates the assertions $\beta_{\text{in}}$ and $\beta_{\text{out}}$ to $\beta_{\text{pre}}$ and $\beta_{\text{post}}$. The representation mapping for stacks takes the record components `S.storage` and `S.top` into `sequences`:

$$A(\ \texttt{S.storage},\ \texttt{S.top}\ ) = \text{seq}(\ \texttt{S.storage}, 1, \texttt{S.top}\ )$$

Using the verification steps shown below, we can demonstrate that package **stack_pkg** is correctly implemented (ignoring initialized variables).

1). Prove that for a concrete object $x$, the concrete invariant $I_c$ implies the abstract invariant $I_a$ (with proper mapping):

$$I_c(x) \supset I_a(A(x))$$

2). Show that the body of each concrete operation $P$ satisfies its concrete input/output specifications $\beta_{\text{in}}$ and $\beta_{\text{out}}$, and maintains $I_c$ for any arguments:

$$\beta_{\text{in}}(x) \land I_c(x) \{ P \} \beta_{\text{out}}(x) \land I_c(x)$$

3). Show that the concrete operation satisfies its abstract input/output specifications:

a). $I_c(x) \land \beta_{\text{pre}}(A(x)) \supset \beta_{\text{in}}(x)$

b). $I_c(x) \land \beta_{\text{pre}}(A(x')) \land \beta_{\text{out}}(x) \supset \beta_{\text{post}}(A(x))$

## 6.2. Proving Programs with Data–Oriented Exception Handling

The operational approach to correctness for modules can be extended to verify a program with data–oriented exception handling. Verification is divided into two parts: proving the correctness of modules containing **#exception** clauses and **#raise** statements and proving user programs containing **#when** clauses.

### 6.2.1. Specifying Pre–Conditions and Post–Conditions for Exceptions

In a package specification, we describe behaviors of exceptions and visible operations exported by the package. The expected behaviors of an exception can be specified abstractly by means of a pair of assertions $E_{\text{pre}}$ and $E_{\text{post}}$, whose roles are similar to those of $\beta_{\text{pre}}$ and $\beta_{\text{post}}$ for visible operations. $E_{\text{pre}}$ represents the conditions that must be satisfied before the exception $E$ is raised, and $E_{\text{post}}$ specifies the conditions that should be true after $E$ is properly handled. For example, in the package specification part of **stack_pkg**, the **#exception** clause attached to the declaration of type **stack** can contain assertions for **overflow** which are specified abstractly in terms of **sequences**:

```
type stack is limited private
    #exception overflow( S : in out stack; place : string ),
            -- Epre  : length(S) = max_size(S) ≥ 0
            -- Epost : 0 ≤ length(S) < max_size(S)
```

### 6.2.2. Supplying Input/Output Assertions for #raise Statements

For each **#raise** statement $R$ in the package body, an input assertion $R_{\text{in}}$ and an output assertion $R_{\text{out}}$ are expressed using terms from the concrete level. $R_{\text{in}}$ specifies the state of the computation before execution of the **#raise** statement, and $R_{\text{out}}$ describes the conditions that should be true when control returns to the point following the **#raise** statement. As an example, the following code segment shows the input/output assertions for a **#raise** statement in the body of procedure **push**:

```
procedure push( S : in out stack;  E : elem_type ) is
begin
    if S.top = S.storage'last then
            -- R_in : S.top = S.storage'last ≥ 0
        #raise overflow( S, "in procedure PUSH" );
            -- R_out :  0 ≤ S.top < S.storage'last
    end if;
    ...
end push;
```

### 6.2.3. Proving #raise Statements in Package Bodies

To demonstrate that a **#raise** statement raises an exception correctly, we need to use a representation mapping $A$ to show that for each **#raise** statement $R$ signaling an exception $E$, $R$'s concrete input assertion $R_{\text{in}}$ implies $E$'s abstract pre-condition $E_{\text{pre}}$ (after proper mapping with $A$). The second proof rule requires that $E$'s abstract post-condition (after proper mapping with $A$) should imply $R$'s concrete output assertion. Thus after the raised exception $E$ has been properly handled, if control returns to the signaler (the operation containing $R$), the expected conditions for resuming the signaler's execution should be satisfied. If control does not return to the signaler, $E_{\text{post}}$ is **false** which always implies $R_{\text{out}}$. More concisely, we need to show that:

$$\frac{R_{in}(\ x\ ) \supset E_{pre}(\ A(x)\ ),\quad E_{post}(\ A(x)\ ) \supset R_{out}(\ x\ )}{R_{in}(\ x\ )\ \{\ \texttt{\#raise}\ E(\ x\ )\ \}\ R_{out}(\ x\ )}$$

where $A(\ x\ )$ is the representation mapping. This mapping is applied to variables in $R_{in}$ and $R_{out}$, before they are substituted for formal parameter names in $E_{pre}$ and $E_{post}$.

For example, to prove that the **#raise** statement in the procedure **push** raises the exception `overflow` on `S` correctly, we need to show:

$$R_{in}(\ \texttt{S}\ ) \supset E_{pre}(\ A(\texttt{S})\ ).$$

The code segment for **push** shows that $R_{in}(\texttt{S})$ is:

$$\texttt{S.top} = \texttt{S.storage'last} \geq 0,$$

and $E_{pre}(\texttt{S})$ for exception `overflow` raised on `S` is:

$$\text{length}(\texttt{S}) = \text{max\_size}(\texttt{S}) \geq 0.$$

After applying the representation mapping for stacks defined in **stack_pkg**:

$$A(\ \texttt{S.storage},\ \texttt{S.top}\ ) = \text{seq}(\ \texttt{S.storage},\ 1,\ \texttt{S.top}\ ).$$

$E_{pre}(A(\texttt{S}))$ is:

$$\text{length}(\ A(\texttt{S})\ ) = \text{max\_size}(\ A(\texttt{S})\ ) \geq 0,$$

$$\text{length}(\ A(\texttt{S.storage},\ \texttt{S.top})\ ) = \text{max\_size}(\ A(\texttt{S.storage},\ \texttt{S.top})\ ) \geq 0,$$

$$\text{length}(\ \text{seq}(\texttt{S.storage},\ 1,\ \texttt{S.top})\ ) = \text{max\_size}(\ \text{seq}(\texttt{S.storage},\ 1,\ \texttt{S.top})\ ) \geq 0.$$

Since $\quad \text{length}(\ \text{seq}(\texttt{S.storage},\ 1,\ \texttt{S.top})\ ) = \texttt{S.top}$

and $\quad \text{max\_size}(\ \text{seq}(\texttt{S.storage},\ 1,\ \texttt{S.top})\ ) = \texttt{S.storage'last},$

$E_{pre}(\ A(\texttt{S})\ )$ becomes: $\quad \texttt{S.top} = \texttt{S.storage'last} \geq 0$

which is exactly $R_{in}(\ \texttt{S}\ )$. Therefore, it is established that

$$R_{in}(\ \texttt{S}\ ) \supset E_{pre}(\ A(\texttt{S}\ )\ ).$$

Similarly, we prove that after a handler associated with **overflow** for S is executed, the assertion following the **#raise** statement in the procedure **push** holds. That is

$$E_{post}(\ A(\mathtt{S})\ )\ \supset\ R_{out}(\ \mathtt{S}\ )$$

$E_{post}(A(\mathtt{S}))$ is:

$$0 \le \text{length}(\ A(\mathtt{S})\ ) < \text{max\_size}(\ A(\mathtt{S})\ ),$$

$$0 \le \text{length}(\ A(\mathtt{S.storage},\ \mathtt{S.top})\ ) < \text{max\_size}(\ A(\mathtt{S.storage},\ \mathtt{S.top})\ ),$$

$$0 \le \text{length}(\ \text{seq}(\mathtt{S.storage},\ 1,\ \mathtt{S.top})\ ) < \text{max\_size}(\ \text{seq}(\mathtt{S.storage},\ 1,\ \mathtt{S.top})\ ),$$

Since $\quad \text{length}(\ \text{seq}(\mathtt{S.storage},\ 1,\ \mathtt{S.top})\ ) = \mathtt{S.top}$

and $\quad \text{max\_size}(\ \text{seq}(\mathtt{S.storage},\ 1,\ \mathtt{S.top})\ ) = \mathtt{S.storage'last},$

$E_{post}(\ A(\mathtt{S})\ )$ becomes: $\quad 0 \le \mathtt{S.top} < \mathtt{S.storage'last}$

which is exactly $R_{out}(\ \mathtt{S}\ )$.

According to the proof rule, the truth of the premises

$$R_{in}(\ \mathtt{S}\ )\ \supset\ E_{pre}(\ A(\mathtt{S}\ )\ ),\qquad E_{post}(\ A(\mathtt{S})\ )\ \supset\ R_{out}(\ \mathtt{S}\ )$$

leads to the conclusion $R_{in}(\ \mathtt{S}\ )$ { #raise $E(\ \mathtt{S}\ )$ } $R_{out}(\ \mathtt{S}\ )$, which is:

```
            S.top = S.storage'last ≥ 0
{  #raise overflow( S, “in procedure PUSH” )  }
            0 ≤ S.top < S.storage'last
```

### 6.2.4. Proving #when Clauses in User Programs

We also need to verify that programs using the package handle exceptions correctly. Users of a package understand its operations and exceptions in terms of values from the abstract domain via the information in each operation's $\beta_{pre}$ and $\beta_{post}$ and each exception's $E_{pre}$ and $E_{post}$. In a **#when** clause, the semantics of a handler associated with an exception should conform to the abstract specifications of the exception.

To prove that handler $H$ processes exception $E$ correctly, we need to associate an input assertion $H_{in}$ and output assertion $H_{out}$ with the handler in a **#when** clause. Two proof steps are necessary to complete the verification. The first step confirms that the pre-conditions $E_{pre}$ establish the truth of $H_{in}$, and the second step requires that the output assertion $H_{out}$ implies the post-condition $E_{post}$. The proof rule for the declaration of an uninitialized variable [McGettrick 82] has been modified as follows:

$$\frac{\begin{array}{c} E_{pre}(\ x\ ) \supset H_{in}(\ x\ ),\ \ H_{out}(\ x\ ) \supset E_{post}(\ x\ ),\ \ H_{in}(\ x\ )\ \{\ H(\ x\ )\ \}\ H_{out}(\ x\ ), \\ Z\ |\!-\ \ \mathrm{P} \wedge \mathrm{X\#} = \mathit{undefined}\ \ \{\ \mathtt{declare}\ \mathrm{D(X\#/X)}\ \mathtt{begin}\ \mathrm{S(X\#/X)}\ \mathtt{end}\ \}\ \ \mathrm{Q} \end{array}}{\mathrm{P}\ \ \{\ \mathtt{declare}\ \mathrm{X} : \mathrm{T}\ \mathtt{\#when}\ E(\ x\ ) \Rightarrow H(\ x\ );\ \mathrm{D}\ \mathtt{begin}\ \mathrm{S}\ \mathtt{end}\ \}\ \ \mathrm{Q}}$$

Here $Z$ contains the proof obligations stated in Section 6.1 for the type declaration. X# denotes a unique identifier, and the notation $P$(X#/X) denotes systematically substituting X# for all free occurrences of X in $P$. D represents a list of declarations, and S represents a list of statements. $x$ denotes formal parameters of an exception, a handler, or a predicate. A similar proof rule can also be given for variable declarations with initial values.

Proofs are carried out with objects and values at the abstract level. If an object has no handler associated with an exception $E$ defined on its type, the program is terminated if $E$ is raised. Thus we can assume a default handler **abort** is associated with $E$, and that the pre-condition and post-condition for **abort** are **true** and **false**, respectively. Since

$$\begin{array}{ll} E_{pre}(\ x\ ) & \supset \mathtt{true} \\ \mathtt{false} & \supset E_{out}(\ x\ ) \end{array}$$

are always valid, it is never unsafe to associate a handler **abort** with an exception $E$.

We now give an example showing how to verify the correctness of a **#when** clause. Consider the following object declaration:

```
S3 : real_stack.stack
       #when overflow( Stk : in out stack ) => expand( Stk, 100 );
```

where **expand** is a procedure declared in the package **stack_pkg** with the following input/output assertions:

```
procedure expand( S : in out stack; amount : positive );
       -- β_pre  : max_size(S) = M > 0 Λ length(S) = L Λ L ≤ M
       -- β_post : max_size(S) = M * (1 + amount/100) > 0 Λ
                                         length(S) = L Λ L ≤ M
```

Substituting the actual parameters of **expand** in the **#when** clause for the formal parameters in $\beta_{pre}$ and $\beta_{post}$ above results in:

$H_{in}$ : max_size(**Stk**) $= M > 0 \wedge$ length(**Stk**) $= L \wedge L \leq M$
$H_{out}$ : max_size(**Stk**) $= M * (1 + 100/100) > 0 \wedge$ length(**Stk**) $= L \wedge L \leq M$

For exception **overflow** on type **stack**, the pre-conditions and post-conditions (after parameter substitutions) are:

$E_{pre}$ : length(**Stk**) = max_size(**Stk**) $\geq 0$

$E_{post}$ : $0 \leq$ length(**Stk**) < max_size(**Stk**)

$E_{pre}$ implies $H_{in}$:

length(**Stk**) = max_size(**Stk**) $\geq 0 \supset$

max_size(**Stk**) $= M > 0 \wedge$ length(**Stk**) $= L \wedge L \leq M$,

$H_{out}$ implies $E_{post}$:

max_size(**Stk**) $= M * 2 > 0 \wedge$ length(**Stk**) $= L \wedge L \leq M \supset$

$0 \leq$ length(**Stk**) < max_size(**Stk**) because $0 < M < (2 * M)$.

The final step $H_{in}$ { $H$ } $H_{out}$ is obviously true since $H$ is **expand(Stk,100)**. $H_{in}$ is

**expand**'s pre–condition, and $H_{out}$ is **expand**'s post–condition.

## 6.3. Proving Programs with Control–Oriented Exception Handling

Control–oriented exception handling mechanisms hinder verification because they introduce multiple exit points from operations and permit exceptions to be propagated[Luckham 80][Cristian 84]. In [Luckham 80] an exception is not allowed to propagate outside of its scope in an Ada program. Otherwise, it may be impossible to verify a program since the number and nature of exceptions that are propagated to a piece of code are unpredictable. Even with this restriction, proof obligations for operations increase faster than the number of exceptions propagated.

Exceptions that are propagated out of a subprogram are specified in the subprogram's heading. In the following example, exception E is propagated from procedure p. Callers may assume that if E is propagated, then assertion A holds.

```
procedure p( ... ) propagate E assert A is
begin
    begin
        ...
        raise E;     -- B is associated assertion
        ...
    exception
        when E assert B => ... ;
    end;
    ...
    raise E;         -- A is associated assertion
    ...
end p;
```

The meaning of raise $E_i$ is described by the raise axiom:

$$A_i \; \{ \texttt{raise } E_i \} \; \texttt{false}.$$

The meaning of a block is specified by the following axiom:

$$\frac{P \{ S_0 \} Q, \ B_i \{ S_i \} Q}{P \ \{ \textbf{begin } S_0 \textbf{ exception } E_i \textbf{ assert } B_i \Rightarrow S_i \textbf{ end} \} \ Q}$$

Assume p is a procedure with $f_i$, $f_o$ and $f_{io}$ as formal **in**, **out**, and **in out** parameters, respectively, and that the correctness of the body of p has been established with respect to the input condition $I(f_i, f_{io})$ and output condition $O(f_i, f_o, f_{io})$. Let $a_i$, $a_o$, and $a_{io}$ be the corresponding actual parameters of a call to p; then this call is described by the rule:

$$\frac{\begin{array}{c} P \supset I(a_i, a_{io}) \wedge \forall a_o, a_{io.} (O(a_i, a_o, a_{io}) \supset Q) \\ P \supset I(a_i, a_{io}) \wedge \forall a_o, a_{io.} (A_1(a_i, a_o, a_{io}) \supset B_1) \\ \ldots \\ P \supset I(a_i, a_{io}) \wedge \forall a_o, a_{io.} (A_n(a_i, a_o, a_{io}) \supset B_n) \end{array}}{P \ \{ \mathrm{p}( a_i, a_o, a_{io} ) \} \ Q}$$

where the clause $P \supset I(a_i, a_{io}) \wedge \forall a_o, a_{io.} (A_j(a_i, a_o, a_{io}) \supset B_j)$ has to be proved for each propagated exception $E_j$ with assertion $A_j$ appearing in the procedure header and handler pre-condition $B_j$ associated with $E_j$ in the calling environment.

Consider the effort needed to verify a program when a new exception $E_{n+1}$ is raised in an operation p of type T. In either exception handling method, the new body of the operation needs to be verified. For control-oriented exceptions, the worst case for re-verification occurs when the exception is propagated, the operation's specifications change, and handlers are added to each block containing an invocation of p. Each invocation of the operation needs to have an additional premise discharged:

$$P \supset I(a_i, a_{io}) \wedge \forall a_o, a_{io.} (A_{n+1}(a_i, a_o, a_{io}) \supset B_{n+1})$$

Also, we need to demonstrate that each new handler body establishes the post-condition of the block in which it is embedded.

For data-oriented exceptions, the worst case occurs when every variable of type `T` is declared in a separate statement and that each declaration contains a **#when** clause for $E_{n+1}$. The three proof obligations for the declaration rule must be carried out for each object. Since there are generally more operation invocations than declared objects in a program, it may be easier to verify data-oriented exception handling programs than their control-oriented exception handling counterparts when programs change.

# CHAPTER 7

# Testing

Most programs are simply too big and complex to be verified. Thus programmers resort to traditional testing methods to increase their confidence about their software. This chapter introduces a simple structural test coverage metric for exception handling. We compare the test cases needed to satisfy this metric for a (slightly simplified) version of `compord.src` found in the Simtel20 Ada Repository with the data needed to test another version of the program using data–oriented exception handling. Finally, we show how to build a tool based on the metric to assist users in constructing better test cases.

## 7.1. A Structural Coverage Metric for Testing Exception Handling

During program testing, a set of test cases is constructed and the program is executed. For each test case, results are checked against a specification (generally an input/output pair) to detect any inconsistencies. Since only a relatively small number of the possible test cases can be executed it is natural to ask how representative the test cases selected are. Structural test coverage metrics are used to measure how well a set of test cases exercise particular program units, e.g., the percentage of statements tested or branches followed. Programmers need to provide enough test data to justify the structure of their programs. For example, if a particular statement is not executed by any test data, either the statement is unreachable or the test data is deficient. Often, structural coverage metrics help programmers discover errors by testing their code more thoroughly than they would have otherwise.

Our structural coverage metric measures the percentage of all explicit signaler–handler bindings exercised during program testing. We define an *ESH-set* as a set whose elements are triples $(e,s,h)$ where $e$ is the name of an exception; $s$ is the statement number of a signaler for the exception $e$ (either a `raise` statement or a procedure invocation containing a `raise` statement); and $h$ is the statement number of a handler invoked to deal with the exception $e$ raised by $s$. For example, the following Ada code segment

```
    procedure P( A, B : integer ) is
        N : natural;
    begin
1:      N := A * B;     -- can raise numeric_error or constraint_error
        ...
5:      N := B;         -- can raise constraint_error
        ...
    exception
        when numeric_error =>
8:          H1;
        when constraint_error =>
9:          H2;
    end P;
```

has two signalers (statements 1 and 5), and two handlers `H1`, `H2`. The program's ESH-set is:

`{ (numeric_error,1,8), (constraint_error,1,9), (constraint_error,5,9) }`

This metric measures more than statement coverage. Consider the case where statement 1 is executed and raises `numeric_error` and (on a subsequent invocation of `P`) statement 5 is executed and raises `constraint_error`. Statements 1, 5, 8, 9 are all executed; however, the binding represented by the triple `(constraint_error,1,9)` in the ESH-set is not exercised.

## 7.2. A Case Study

Our structural coverage metric can be used to determine the effort required to test a program by considering the number of test cases needed to satisfy the metric for the program. We again examine the procedure put_info_in_dag that utilizes an abstract data type dag specified in the file abstract.src:

```
with set_pkg; ...
generic
    type label  is private;
    type value is private;
    ...
package dag_pkg is
    type dag is private;

    illegal_node: exception;
    duplicate_edge: exception;
    makes_cycle: exception;

    procedure add_node( g: in out dag; l: in label; v: in value );
            -- raise illegal_node if the node is already in the dag

    procedure add_edge( g: in out dag; l1: in label; l2: in label );
            -- may raise duplicate_edge or makes_cycle

    procedure set_value( g: in out dag; l: in label; v: in value );
            -- raise illegal_node if the node is not in the dag

    function  get_value( g: dag; l: label ) return value;
            -- raise illegal_node if the node is not in the dag
    ...
end dag_pkg;
```

In the file compord.src, several package specifications are referenced by the procedure put_info_in_dag:

```
with dag_pkg; with nodes; ...
package units_dag_pkg is new dag_pkg ( ... , dag_node, ... );

with dag_pkg; with nodes; ...
package mini_dag_pkg is new dag_pkg ( ... , empty_node, ... );

with units_dag_pkg; with mini_dag_pkg; ...
package compiler_order_declarations is

    subtype units_dag is units_dag_pkg.dag;
    subtype info_dag  is  mini_dag_pkg.dag;

    withs_dag: units_dag := units_dag_pkg.create;
    files_dag: info_dag  := mini_dag_pkg.create;
    cycle_dag: info_dag  := mini_dag_pkg.create;
    ...
end compiler_order_declarations;
```

`put_info_in_dag` is a procedure in the body of package `compile_order_utilities` which puts information about compilation dependencies of Ada program units into `withs_dag`. If a new edge added to `withs_dag` results in a cycle, the newly added edge and nodes are entered into `cycle_dag` for later error reporting.

```
with compile_order_declarations; ...
package body compile_order_utilities is
   ...
   package COD renames compile_order_declarations;
   package WDAG renames units_dag_pkg;     -- WDAG for withs dag
   package IDAG renames mini_dag_pkg;      -- IDAG for info  dag
   ...
   procedure put_info_in_dag( node_label : in SP.string_type;
                              info_list : in out COD.id_list_pkg.list ) is
      i : COD.id_list_pkg.listiter;
      with_node : SP.string_type;
      with_name : SP.string_type;
      label     : SP.string_type;
      value     : nodes.dag_node;
      gen_inst  : boolean;
   begin
      ...
      begin
         label  := SP.upper(node_label);
         value := WDAG.get_value(COD.withs_dag, label);
         gen_inst := SP.equal(COD.current_file, "");
```

```
      ...
   if not gen_inst then
      ...
      WDAG.set_value(COD.withs_dag, label, value);
   end if;
exception
   when WDAG.illegal_node =>
      -- the node doesn't exist yet so we must add it
      value := COD.default_node;
      value.file := SP.make_persistent(COD.current_file);
      value.name := SP.make_persistent(node_label);
      WDAG.add_node(withs_dag,
                   SP.make_persistent (label), value);
end;
i := COD.id_list_pkg.MakeListIter(info_list);
while COD.id_list_pkg.more(i) loop
   COD.id_list_pkg.next(i, with_name);
   begin
      ...
      WDAG.add_node(COD.withs_dag, with_node, value);
      WDAG.add_edge(COD.withs_dag, label, with_node);
   exception
      when WDAG.illegal_node =>
         -- Raised when the with_node is already in
         -- the dag. No harm done so ignore the error
         -- and add the edge.
         begin
            WDAG.add_edge(COD.withs_dag, label, with_node);
         exception
            when WDAG.makes_cycle =>
               begin
                  IDAG.add_node(COD.cycle_dag,
                              SP.make_persistent(label),
                              COD.default_empty_node);
                  IDAG.add_node(COD.cycle_dag,
                              SP.make_persistent(with_node),
                              COD.default_empty_node);
                  IDAG.add_edge(COD.cycle_dag, label,with_node);
               exception
                  when IDAG.illegal_node | IDAG.makes_cycle =>
                               null;
               end;
            when WDAG.duplicate_edge =>
                         null;
         end;
      when WDAG.makes_cycle =>
         -- need to keep track of where the cycles are.
         begin
```

```
                IDAG.add_node(COD.cycle_dag,
                              SP.make_persistent (label),
                              COD.default_empty_node);
                IDAG.add_node(COD.cycle_dag,
                              SP.make_persistent (with_node),
                              COD.default_empty_node);
                IDAG.add_edge(COD.cycle_dag, label, with_node);
             exception
                when IDAG.illegal_node | IDAG.makes_cycle =>
                               null;
             end;
          end;
       end loop;
       ...
    end put_info_in_dag;
 end compile_order_utilities;
```

### 7.2.1. Build_dag with Control-Oriented Exception Handling

The original program is very complex. In addition to more than 2,000 lines of code in **compord.src**, the program uses about 13,900 lines of code in **abstract.src** containing a variety of generic packages implementing about two dozen data types. We built a simplified version of the program that maintains the original algorithms and control logic in **put_info_in_dag**, but disregards other unrelated activities. The main procedure **build_dag** is:

```
     with generic_dag_pkg;
     procedure build_dag is
        package units_dag_pkg is new generic_dag_pkg( character );
        use units_dag_pkg;

        withs_dag, cycle_dag : dag;

        procedure add_to_cycle_dag( parent_node, with_node : character )
                                                           is separate;
            -- may raise duplicate_node, duplicate_edge, makes_cycle

        procedure put_info_in_dag( parent_node, with_node : character )
                                                           is separate;
     begin
  1:    withs_dag := create;
```

```
 2:    cycle_dag := create;

 3:    put_info_in_dag( 'A', 'A' );    -- with A; package A is ...
 4:    put_info_in_dag( 'A', 'B' );    -- with B; package A is ...
 5:    put_info_in_dag( 'A', 'B' );    -- with B; package A is ...
 6:    put_info_in_dag( 'A', 'A' );    -- with A; package A is ...
    end build_dag;

    --------------------------------------------------------------

    separate( build_dag )
    procedure put_info_in_dag( parent_node, with_node : character ) is
    begin
 7:    begin
 8:       check_node( withs_dag, parent_node );
       exception
          when node_not_in_dag =>
 9:           add_node( withs_dag, parent_node );
       end;
10:    begin
11:       add_node( withs_dag, with_node);
12:       add_edge( withs_dag, parent_node, with_node );
       exception
          when duplicate_node =>
13:          begin
14:             add_edge( withs_dag, parent_node, with_node );
             exception
                when makes_cycle =>
15:                begin
16:                   add_to_cycle_dag( parent_node, with_node);
                   exception
                      when duplicate_node | makes_cycle =>
17:                          null;
                   end;
                when duplicate_edge =>
18:                       null;
             end;
          when makes_cycle =>
19:          begin
20:             add_to_cycle_dag( parent_node, with_node);
             exception
                when duplicate_node | makes_cycle =>
21:                    null;
             end;
          when duplicate_edge =>
22:              null;
       end;
    end put_info_in_dag;
```

---

```
    separate( build_dag )
    procedure add_to_cycle_dag( parent_node, with_node : character ) is
    begin
23:    add_node( cycle_dag, parent_node);
24:    add_node( cycle_dag, with_node);
25:    add_edge( cycle_dag, parent_node, with_node);
    end add_to_cycle_dag;
```

Comparing the program shown above to the original version, we see that exceptions raised and handled in the original procedure are also raised and handled similarly in the simplified version. An ESH–set for this program can be constructed by studying the control flow and exception handling behavior of the program. In a later section, we will present an algorithm for building this set automatically. The ESH–set contains the following 12 (e,s,h) triples:

```
{ (node_not_in_dag,  8, 9),   (duplicate_node, 11, 13),
  (makes_cycle, 12, 19),      (duplicate_edge, 12, 22),
  (makes_cycle, 14, 15),      (duplicate_edge, 14, 18),
  (duplicate_node, 23, 17),   (duplicate_node, 24, 17),
  (makes_cycle, 25, 17),      (duplicate_node, 23, 21),
  (duplicate_node, 24, 21),   (makes_cycle, 25, 21)      }
```

Statements 3 through 6 constitute a set of test cases for testing the procedure **put_info_in_dag**. The particular test case set was chosen because it is a minimum test cases needed to cover all accessible triples. Statement 3 attempts to add edge ('A','A') to **withs_dag**, raising exceptions in the following statements. Statement 8 raises **node_not_in_dag** when it finds **withs_dag** has no nodes; statement 11 raises **duplicate_node** when it tries to add **with_node** (whose value is 'A'); and statement 14 raises **makes_cycle** when adding the edge ('A','A'). **add_to_cycle_dag** is then invoked to add **parent_node**, **with_node**, and the edge ('A','A') to **cycle_dag**. Again,

**duplicate_node** is raised when adding **with_node** to **cycle_dag** and the invocation to **put_info_in_dag** is terminated. After the second test case (statement 4) successfully adds a new node 'B' and an edge ('A','B') to **withs_dag**, statement 5 causes **duplicate_edge** to be raised when trying to add the same edge to **withs_dag**. The final test case (statement 6) causes **duplicate_node** to be raised when trying to add 'A' as **parent_node** to **cycle_dag**.

Any test set with less than four test cases fails to cover some part of the reachable code. Statement 14 (**add_edge**) cannot raise **duplicate_edge** and **makes_cycle** at the same time. Two invocations of **put_info_in_dag** (statements 3 and 6) that cause statement 14 to raise **makes_cycle** for **withs_dag** are needed in order for statement 23 (**add_node**) to raise **duplicate_node** on **cycle_dag**. Since statements 3 and 6 do not add any edges to **withs_dag** because ('A','A') would be a cycle, two additional executions of **put_info_in_dag** (statements 4 and 5) that do not raise **makes_cycle** are needed to cause statement 14 to raise **duplicate_edge**. Thus, a test data set exercising all reachable (e,s,h) triples for this program should contain at least four invocations to **put_info_in_dag**.

Table 7 shows the (e,s,h) triples exercised by executing these statements. The execution of the test cases leaves six triples in the ESH-set:

```
(makes_cycle, 12, 19),      (duplicate_edge, 12, 22),
(duplicate_node, 23, 21),  (makes_cycle, 25, 17),
(duplicate_node, 24, 21),  (makes_cycle, 25, 21)
```

No extra test data exercises these bindings because the code associated with these triples is unreachable. For example, the four triples:

| Statement | ESH triples exercised |
|---|---|
| 3 | (node_not_in_dag, 8, 9), (duplicate_node, 11, 13), (makes_cycle, 14, 15), (duplicate_node, 24, 17) |
| 4 | |
| 5 | (duplicate_node, 11, 13), (duplicate_edge, 14, 18) |
| 6 | (duplicate_node, 11, 13), (makes_cycle, 14, 15), (duplicate_node, 23, 17) |

**Table 7. ESH Triples Exercised by Executing Build_Dag ( Control )**

```
(makes_cycle, 12, 19),     (duplicate_node, 23, 21),
(duplicate_node, 24, 21), (makes_cycle, 25, 21)
```

are associated with the handler for `makes_cycle` starting from statement 19. In order for control to reach this `begin`-block, the procedure invocation `add_edge` (statement 12) has to raise the exception `make_cycle`. This implies that statement 11 (the procedure invocation `add_node`) must be completed without raising any exceptions. In such a situation, `with_node` is a fresh node just added into `withs_dag`. Note that `with_node` must be a node different from `parent_node`, otherwise `duplicate_node` is raised by statement 11. Since there is no edge connected to `with_node` yet and `with_node` is different from `parent_node`, adding an edge from `parent_node` to `with_node` will never cause a cycle. A similar argument holds for the `null` handler (statement 22) associated with the exception `duplicate_edge`. Therefore, triple (`duplicate_edge, 12, 22`) is unreachable. The triple (`makes_cycle, 25, 17`) is also unreachable since `makes_cycle` is not raised at statement 25 unless the two preceding **add_node** statements (23 and 24) have executed without raising any exceptions. Thus **parent_node** and **with_node** must be new, distinct nodes in **cycle_dag** so adding an edge from **parent_node** to **with_node** in **cycle_dag** never results in a cycle.

### 7.2.2. Build_dag with Data–Oriented Exception Handling

To evaluate the effect of our exception handling mechanism on testing, we produced another version of **put_info_in_dag** with data–oriented exception handling. The specification of **generic_dag_pkg** with data–oriented exception handling is:

```
generic
   type label is private;      -- labels of nodes
package generic_dag_pkg is
   type dag is private;
         #exception duplicate_node( g : in out dag; l : label ),
                    node_not_in_dag( g : in out dag; l : label ),
                    duplicate_edge( g : in out dag; l1, l2 : label ),
                    makes_cycle( g : in out dag; l1, l2 : label);

   function  create return dag;

   procedure add_node( g : in out dag; l : label );
           -- may raise duplicate_node

   procedure add_edge( g : in out dag; l1, l2 : label );
           -- may raise duplicate_edge, makes_cycle

   procedure check_node( g : dag; l : label );
           -- may raise node_not_in_dag
private
   type dag_object;
   type dag is access dag_object;
end generic_dag_pkg;
```

The main procedure **build_dag** is:

```
with generic_dag_pkg;
procedure build_dag is

   package units_dag_pkg is new generic_dag_pkg( character );
```

```
     use units_dag_pkg;

     procedure add_to_cycle_dag( parent_node, with_node : character );

     withs_dag : dag;
       #when node_not_in_dag( d : in out dag; node : label ) =>
 1:                                            add_node( d, node ),
             duplicate_node( g : in out dag; l : label ) =>
 2:                                                        null,
             duplicate_edge( g : in out dag; l1, l2 : label ) =>
 3:                                                           null,
             makes_cycle( g : in out dag; l1, l2 : label ) =>
 4:                                       add_to_cycle_dag( l1, l2 );
     cycle_dag : dag;
       #when duplicate_node( g : in out dag; l : label ) =>
 5:                                                        null,
             makes_cycle( g : in out dag; l1, l2 : label ) =>
 6:                                                        null;

     procedure put_info_in_dag( parent_node, with_node : character )
                                                        is separate;
     procedure add_to_cycle_dag( parent_node, with_node : character )
                                                        is separate;
   begin
 7:  withs_dag := create;
 8:  cycle_dag := create;

 9:  put_info_in_dag( 'A', 'A' );   -- with A; package A is ...
10:  put_info_in_dag( 'A', 'B' );   -- with B; package A is ...
11:  put_info_in_dag( 'A', 'B' );   -- with B; package A is ...
12:  put_info_in_dag( 'A', 'A' );   -- with A; package A is ...
   end build_dag;

   ------------------------------------------------------------------

   separate( build_dag )
   procedure put_info_in_dag( parent_node, with_node : character ) is
   begin
13:  check_node( withs_dag, parent_node );
14:  add_node( withs_dag, with_node);
15:  add_edge( withs_dag, parent_node, with_node );
   end put_info_in_dag;

   ------------------------------------------------------------------

   separate( build_dag )
   procedure add_to_cycle_dag( parent_node, with_node : character ) is
   begin
```

```
16:     add_node( cycle_dag, parent_node);
17:     add_node( cycle_dag, with_node);
18:     add_edge( cycle_dag, with_node);
     end add_to_cycle_dag;
```

This version is much simpler then the previous version. The procedure **put_info_in_dag** contains only three subprogram invocations because the code for exception handling is associated with exceptions in declarations. The ESH-set contains the following seven (e,s,h) triples:

```
{ (node_not_in_dag, 13, 1), (duplicate_node, 14, 2),
  (makes_cycle, 15, 4),     (duplicate_edge, 15, 3),
  (duplicate_node, 16, 5),  (duplicate_node, 17, 5),
  (makes_cycle, 18, 6)                               }
```

| Statement | ESH triples exercised |
|---|---|
| 9 | (node_not_in_dag, 13, 1), (duplicate_node, 14, 2), (makes_cycle, 15, 4), (duplicate_node, 17, 5), (makes_cycle, 18, 6) |
| 10 | |
| 11 | (duplicate_node, 14, 2), (duplicate_edge, 15, 3) |
| 12 | (duplicate_node, 14, 2), (makes_cycle, 15, 4), (duplicate_node, 16, 5), (duplicate_node, 17, 5), (makes_cycle, 18, 6) |

**Table 8. ESH Triples Exercised by Executing Build_Dag ( Data )**

Table 8 lists the (e, s, h) triples exercised by each of the four test cases (statements 9 through 12); the test data covers all triples in the ESH–set. The ESH–set for `build_dag` with control–oriented exception handling is larger than that for `build_dag` with data–oriented exception handling. The former contains six unreachable (e,s,h) triples, while all of the triples in the latter can be accessed. Note that (`makes_cycle,18,6`) in the latter corresponds to an unreachable triple (`makes_cycle,25,17`) in the former. Except this slight difference, the two versions of `build_dag` have the same logic, therefore both versions need the minimum of four operation invocations to cover all reachable (e,s,h) triples.

The complexity of `build_dag` with control–oriented exception handling makes it more difficult to test the program when the number of exceptions that can be raised grows. Assume a new exception `uninitialized_dag` can be raised by `check_node`, `add_node`, and `add_edge`. If this exception is always handled by a signaler's immediate invoker, 11 more (e,s,h) triples are added to the ESH–set for `build_dag` with control–oriented exception handling. When statements 8, 9, 11, 12, 14 raise the exception, it is handled in the main program `build_dag`; and when statements 23, 24, 25 raise the exception, it is handled in blocks 15 and 19 respectively. Even after the unreachable code in the control–oriented version is removed, there are still eight more (e,s,h) triples added to the ESH–set. By way of comparison, in the data–oriented exception handling version, only 6 (e,s,h) triples are added to the ESH–set.

### 7.3. Automated Coverage Metric Evaluation

A two–phase pre–processor can be built to assess structural coverage automatically. The first phase constructs an ESH–set for the program, while the second uses the ESH–

set to insert diagnostic code into the user program. Upon finishing execution, the transformed program reports the (e,s,h) triples (if any) that have not been exercised during the program's execution.

### 7.3.1. Algorithms for Constructing ESH–Sets

In this section, we present algorithms for constructing ESH–sets. Two algorithms are introduced: one constructs ESH–sets for Ada programs and the other builds ESH–set for pseudo–Ada programs with data–oriented exception handling. The first algorithm uses a static call graph to analyze the subprogram invocation dependencies in an Ada program and attaches nodes representing exception handlers to nodes for subprograms or blocks. An (e,s,h) triple is associated with an edge from a subprogram or block node to a handler node. Once the whole program is processed, the graph is traversed and (e,s,h) triples are collected to build the ESH–set. The algorithm consists of the following four steps:

Step 1: Constructing a call graph. A call graph describing the subprogram invocation dependencies in the source program is created. Blocks are treated as anonymous subprograms declared and invoked at the same place. Each node in the graph shows the name of the subprogram called (or the name `block` if the node represents a block). In addition, each node is labeled with a unique statement number to distinguish different invocations of a subprogram. For example, a call graph shown in Figure 2 can be derived from the following program skeleton:[†]

[†] For simplicity, consecutive numbers are used to label statements in different modules in our examples. As a more realistic statement labeling scheme, pairs in the form of (module_name, offset) can be used to label statements to suit the needs of separate compilation. For example, (main, 1), (main, 2), ..., (P1, 1), (P1, 2), ...

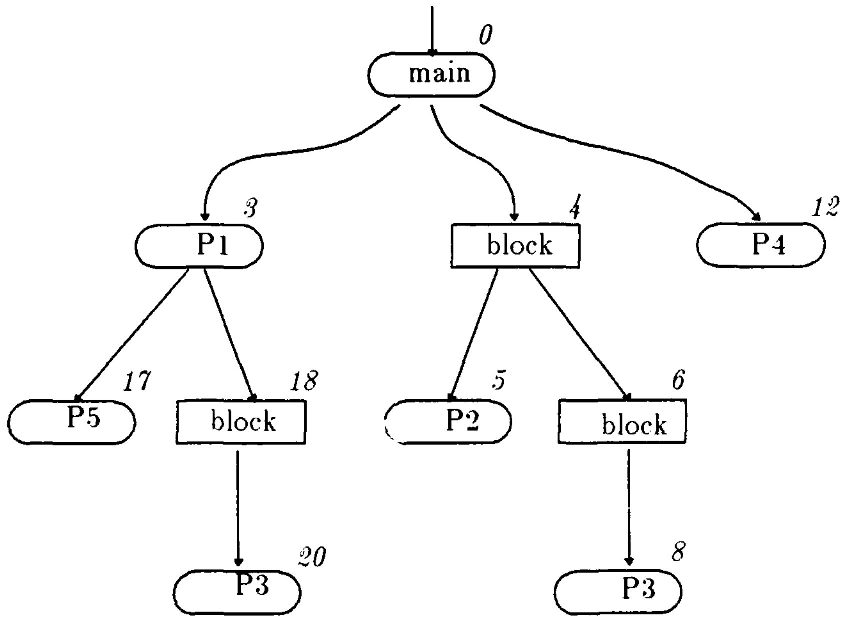



**Figure 2. Call Graph of a Sample Program**

```
      procedure main is
         procedure P1( ... )
                 is separate;
         ...
      begin
         ...
3:       P1( ... );
4:       begin
5:          P2( ... );
6:          begin
               ...
8:             P3( ... );
            end;
            ...
         end;
12:      P4( ... );
      exception
         when E2 =>
13:            P2( ... );
      end main;
```

```
      separate( main )
      procedure P1( ... ) is
      begin
         ...
15:      if B then
16:         raise E1;
         end if;
17:      P5( ... );
18:      begin
            ...
20:         P3( ... );
         exception
            ...
         end;
      exception
         when E1 =>
23:            P6( ... );
24:            P7( ... );
         when E2 =>
               ...
      end P1;
```

In the graphic representation of the call graph, we use ovals to represent subprograms and rectangles for blocks, although these nodes are treated uniformly in our algorithm.

Step 2: Attaching handler nodes to subprogram/block nodes. For each subprogram or block node N in the call graph, if the subprogram or block contains a **raise** statement for exception E, perform one of the following two actions:

Step 2.1: If a handler H associated with E exists in the current subprogram or block, attach a handler node $N_h$ as a child node of N. $N_h$ is labeled with the statement number of the first statement in the handler H. Any subprogram invocation or block statement in H becomes a child node of $N_h$. Such a child node is in fact the root node of a sub-graph because there can be further invocations in the subprogram or block. The edge from N to $N_h$ is labeled with a triple $(E,S_1,S_2)$ where $S_1$ is the statement number of N and $S_2$ is the statement number of the first statement in H. As an example, Figure 3 shows a handler node attached to node P1.

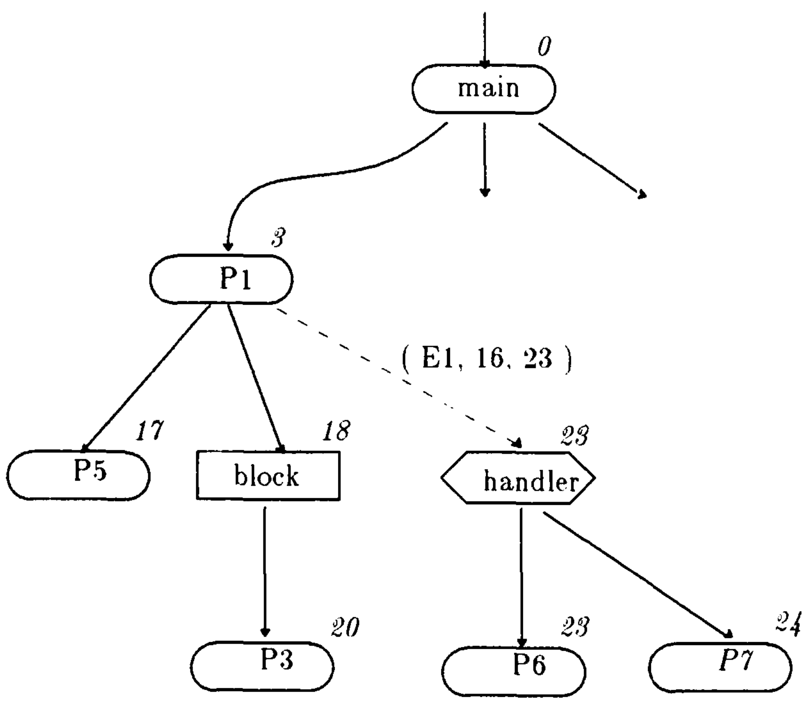



**Figure 3. Attaching a Local Handler Node**

The handler is associated with the `raise` statement (statement 16) and contains two subprogram invocations (statements 23 and 24). Dashed edges are used to distinguish handler associations from subprogram invocations.

Step 2.2: If there is no handler associated with E in the current subprogram or block, move upward along the path from N to the root node until either a subprogram or block node $N_p$ with a handler H for E is found or the root node is passed. If a handler H is found, attach a handler node to $N_p$ as in step 2.1. For example, if in our sample program, `P6` has a `raise` statement signaling the exception `E2` without a corresponding handler associated with `E2` in `P6`, a handler node will be attached to node `main` as in Figure 4.

Step 3: Repeat Step 2 for the new nodes. For each of the new subprogram or block nodes introduced by the Step 2, repeat Step 2 until the graph cannot be expanded.

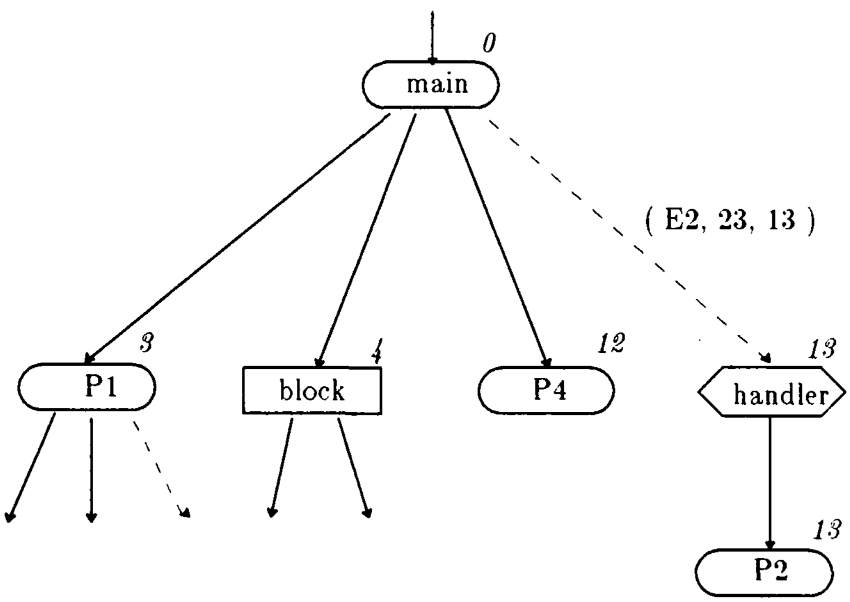



**Figure 4. Attaching a Global Handler Node**

Step 4: Building the ESH-set. Traverse the graph and collect all the (e,s,h) triples attached to edges leading to handler nodes.

Figure 5 shows the call graph with handler nodes for `build_dag` with control-oriented exception handling. The ESH-set for the program can be obtained by collecting the 12 (e,s,h) triples attached to the edges leading to handler nodes in the graph.

The second algorithm constructs ESH-sets for programs with our exception handling mechanism. It consists of the following steps:

Step 1: Constructing a call graph. As in the previous algorithm, a call graph is used to describe the subprogram invocation dependencies in a program. However, blocks are not treated as anonymous subprogram invocations.

Step 2: Attaching handler nodes to subprogram invocation nodes. For each subprogram invoked containing `#raise` statements, collect all the objects on which exceptions are raised. If an object has a handler H defined for an exception, a node $N_h$ representing the handler invocation is inserted to the call graph as a child node of the subprogram invocation node N. A triple (e,s,h) is attached to the edge from N to $N_h$, where e is the exception raised, s is the statement number of the signaler represented by N, and h is the statement number of the handler H.

Step 3: Repeat Step 2 for the new nodes. If the new node $N_h$ added into the graph represents a subprogram invocation, it can be further expanded to a sub-graph if the handler invokes other subprograms.

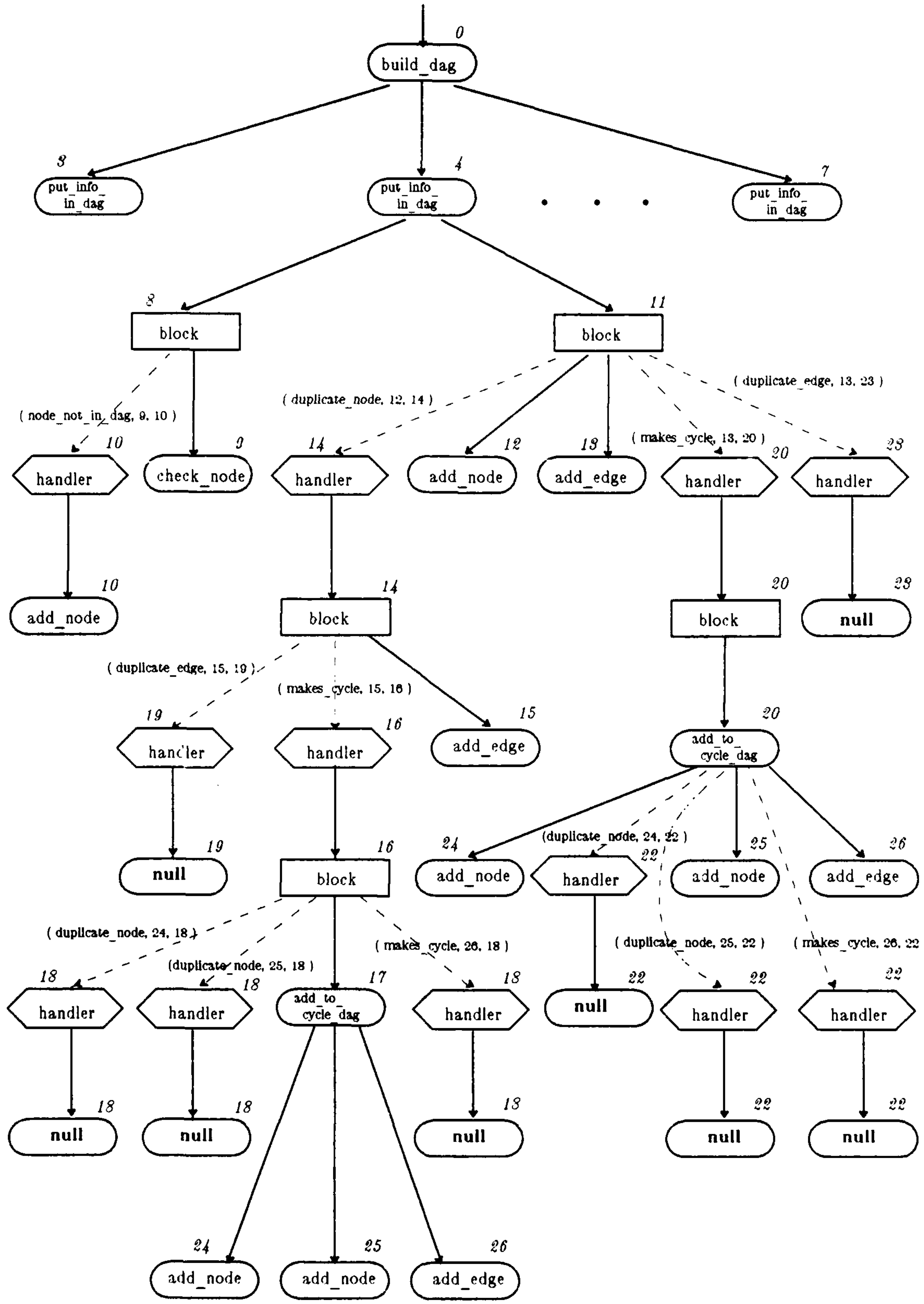



**Figure 5. Call Graph for Build_Dag ( Control–Oriented )**

Step 4: Building the ESH-set. After all the nodes in the call graph have been processed (including the new nodes introduced by handler invocations), the (e,s,h) triples attached to edges leading to handler nodes are collected to form the ESH-set.

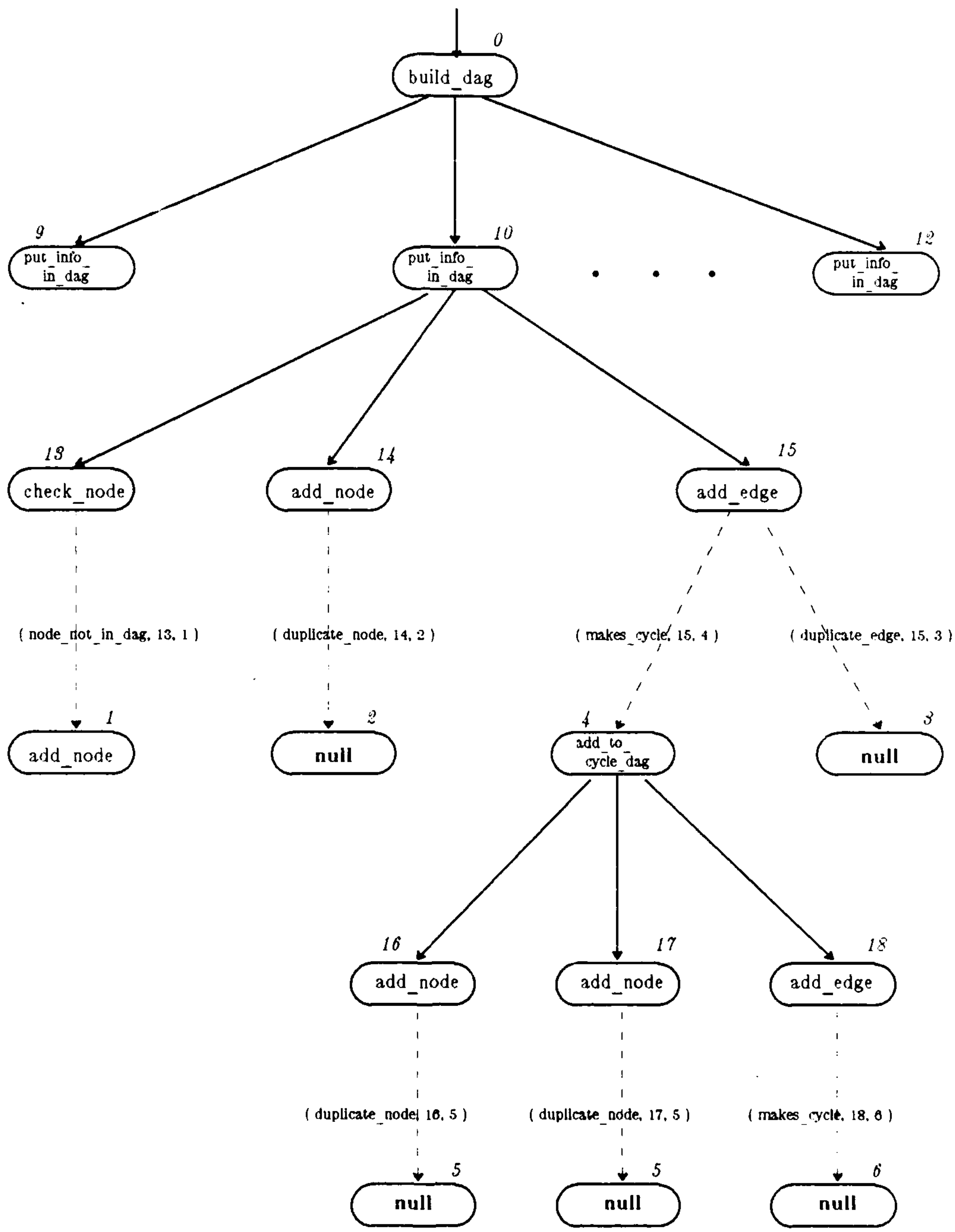



**Figure 6. Call Graph for Build_Dag ( Data-Oriented )**

Figure 6 shows the call graph for `build_dag` with data-oriented exception handling. In the graph, edges with dashed lines represent handler invocations. The seven (e,s,h) triples associated with dashed edges are collected to build the ESH-set for the program.

### 7.3.2. Inserting Diagnostic Code to Source Programs

Once the ESH-set for a program is obtained, it is straightforward for the pre-processor to add diagnostic code to the source program. The inserted code keeps track of the (e,s,h) triples covered in a program execution, and reports those triples that have not been covered (if any) when the execution finishes.

Before the first executable statement in the main program, code is inserted by the pre-processor to initialize a working **esh-set** that contains all the triples in the ESH-set for the program. Before every handler statement, the pre-processor inserts code to remove the corresponding (e,s,h) triple from the working **esh-set**. Finally, the pre-processor appends code after the last executable statement in the main program to report the content of the working **esh-set** after program execution terminates.

A global variable **zzz_stmt_no** is added to the declaration list of the main program; its value is the statement number of potential exception signals. Similarly, a variable **zzz_except_name** global to all compilation units is used to remember the name of an exception raised. The contents of these two variables are used when removing an (e,s,h) triple from the working **esh-set**. Note that in Ada a handler can be associated with more than one exception in a **when** arm, thus it is necessary to remember the name of an exception being raised so that a specific (e,s,h) triple can be singled out.

Places to insert code to remove (e,s,h) triples from the working **esh-set** or to update **zzz_stmt_no** or **zzz_exception_name** can be deduced from the triples in the ESH-set. A statement updating **zzz_except_name** is inserted before a **raise** or **#raise** statement in a package body if the exception name appears in one of the triples in the ESH-set. Similarly, in the main program, a statement updating **zzz_stmt_no** is inserted before a statement whose statement number appears in the second part of an (e,s,h) triple in the ESH-set. Finally, the third part of an (e,s,h) triple in the ESH set determines the place where a statement removing a triple from the working **esh-set** should be inserted in the main program. For example, the following statement

```
zzz_esh_set_pkg.remove(zzz_esh_set, (zzz_except_name, zzz_stmt_no, 10));
```

is inserted immediately before statement 10. The statement removing an (e,s,h) triple from the working **esh-set** is inserted before the handler is transformed and placed in the corresponding dispatch procedure.

### 7.4. Summary

The structural test coverage metric introduced in this chapter helps programmers analyze the less well tested parts of their programs [Horning 79]. In the **build_dag** example with control-oriented exception handling, the six uncovered (e,s,h) triples reveal unreachable code in the program.

Testing programs with control-oriented exception handling tends to be more difficult than testing programs with data-oriented exception handling. The size of the ESH-set for the version with control-oriented exception handling grows faster than that for the version with data-oriented exception handling when additional exceptions are raised. Comparing Figures 5 and 6. we also see that the complexity of control oriented exception handling makes it more difficult to construct an ESH-set for a program with control-oriented exception handling. Figure 6 is simpler than Figure 5 because less complex control flows need to be considered in the process of collecting the (e,s,h) triples.

# CHAPTER 8

# Empirical Studies

We conducted two studies to investigate the effects of different exception handling mechanisms on program construction, comprehension, and modification. The subjects were senior undergraduate students taking an advanced Ada course in the University of Maryland, University College for adult continuing education. All of the students were experienced programmers working for commercial software companies. These studies were performed on relatively small programs written by students to help substantiate claims about benefits provided by data-oriented exception handling. Although the results cannot be generalized to large systems, the data encourages us to apply our methods to these systems.

## 8.1. Program Construction

In our first study, subjects solved the same problem twice, first with Ada and then with our version of Ada with data-oriented exception handling. A pre-processor was provided to translate pseudo-Ada programs.

We tested the following four pairs of null ($H_0$) and alternative ($H_1$) hypotheses:

$H_0$: programs for project 1 and 2 are the same sizes;
$H_1$: programs for project 1 are bigger than programs for project 2.

$H_0$: programs for project 1 and 2 have the same number of statements per subprogram;
$H_1$: programs for project 2 have better modularity (fewer statements per subprogram) than programs for project 1.

$H_0$: the maximum statement nesting depths in programs for projects 1 and 2 are identical.

$H_1$: programs for project 1 have greater maximum statement nesting depths than those for project 2.

$H_0$: the average nesting depth per statement in programs for project 1 is the same as that for project 2;

$H_1$: the average statement nesting depth in programs for project 1 is larger than that for project 2.

Although we evaluate the results of this study as if it were a controlled experiment, we realize that substantial learning effects may bias the results.

Students designed and implemented a generic package supporting an abstract data type **hash_table**. Two hashing functions are supplied when instantiating the package. Initially, the table resorts to linear probing to resolve collisions. If hashing becomes too inefficient due to repeated key collisions, a second function using a relatively complicated algorithm that produces better key distribution is used instead. The size of a hash table should be twice the cardinality of the set of keys to obtain sub-linear search time [Bentley 87]. When it becomes too full, the hash table is expanded at run time. Several exceptions, such as **too_many_nonhit**, **table_half_full** and **no_more_storage**, are declared and raised to facilitate function switching and table expansion. A driver routine tests the exceptions raised at run time and takes appropriate handling actions.

Of the 11 students who remained in the class to the end of the semester, nine of them turned in the programs for both of the projects. Table 7 characterizes the driver programs written by the students. Study of these programs shows that using data-oriented exception handling can result in smaller (and perhaps simpler) code. On average, the driver routines in the first project have 152.67 Ada statements, about 25% more statements than their counterparts in the second project (122.11 statements). On

| Statement and Subprogram Counts in Driver Programs | | | | | | |
|---|---|---|---|---|---|---|
| Account | Project 1 | | | Project 2 | | |
| | Statement Count | Subprogram Count | Statements per Subprogram | Statement Count | Subprogram Count | Statements per Subprogram |
| 05 | 159 | 12 | 13.3 | 142 | 15 | 9.5 |
| 06 | 189 | 7 | 27.0 | 118 | 7 | 16.9 |
| 07 | 148 | 6 | 24.6 | 132 | 9 | 14.7 |
| 11 | 99 | 6 | 16.5 | 109 | 10 | 10.9 |
| 12 | 202 | 8 | 25.3 | 125 | 8 | 15.6 |
| 13 | 150 | 7 | 21.4 | 104 | 7 | 14.9 |
| 14 | 151 | 7 | 21.6 | 116 | 9 | 12.9 |
| 19 | 130 | 5 | 26.0 | 120 | 10 | 12.0 |
| 20 | 146 | 6 | 24.3 | 133 | 9 | 14.8 |
| Average | 152.67 | 7.11 | 22.22 | 122.11 | 9.33 | 13.58 |

**Table 7. Statement and Subprogram Counts in Driver Programs**

average, the subjects divided the driver routines into more sub-modules (procedures and functions) in the data-oriented versions -- 9.33 sub-modules, compared to 7.11 sub-modules used in the control-oriented version. By calculating the ratio of the number of statements to the number of sub-modules in the respective driver routines, we can see that the average size of the procedures/functions in project 2 (13.58 statements/sub-module) is significantly smaller than that in project 1 (22.22 statements/sub-module).

To determine the statistical significance of the difference between two means $\mu_1$ and $\mu_2$, we need to use a nonparametric test such as the *Wilcoxon rank-sum test*[Bhatt 77]. A standard parametric test (e.g., *t-test*) may not be appropriate because we cannot assume our sample data come from a population with normal distribution.

To perform the Wilcoxon rank-sum test on sample population of sizes $m$ and $n$, the values of the two sample populations are ranked jointly, as if they were one sample, in increasing order of magnitude. The values of the joint population are then assigned the ranks 1, 2, ..., m+n. For equal values, each value is assigned the mean of the ranks that the values jointly occupy.

To test the null hypothesis $H_0$ versus the alternative hypothesis $H_1$ for the program sizes, we calculate the following rank sums:

$$W_1 = 16 + 17 + 13 + 1 + 18 + 14 + 15 + 8 + 12 = 114$$

$$W_2 = 11 + 5 + 9 + 3 + 7 + 2 + 4 + 6 + 10 = 57$$

The Wilcoxon's rank-sum statistic table for both sample sizes of nine has the following selected values:

| P[ $W_s \geq x$ ] for both sample sizes = 9 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $x$ | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 |
| $P$ | 0.057 | 0.047 | 0.039 | 0.031 | 0.025 | 0.020 | 0.016 | 0.012 | 0.009 |

Since P[ $W_1 \geq 112$ ] = 0.009, the null hypothesis $H_0$ is rejected at level of significance $\alpha$

$= 0.009$. Similarly, to test the hypotheses for program modularity, the following rank sums are calculated:

$$W_1 = 5 + 18 + 15 + 10 + 16 + 12 + 13 + 17 + 14 = 120$$

$$W_2 = 1 + 11 + 6 + 2 + 9 + 8 + 4 + 3 + 7 = 51$$

Since the reject region with $\alpha = 0.009$ is established as $W_1 \geq 112$ and the observed value falls in this region, the null hypothesis $H_0$ is rejected at level of significance $\alpha = 0.009$.

Other improvements in program structure in the data-oriented versions can be demonstrated by examining the main program part of the driver routine (the top-level executable statement list). Analyzing the counts of executable statements, maximum nesting depths, and average statement nesting depths in the main programs, we obtained the data shown in Table 8. In addition, the means of these three measurements for each project are calculated and presented in the table. On average, there are 36.89 executable statements in the main blocks in project 1, which is more than twice the number in project 2 (16.56 executable statements). Control-oriented exception handling forced subjects to use more deeply nested syntactic structures (5.89 maximum nesting depth and 3.63 per statement in project 1, compared to 3.22 and 2.00 in project 2.) To determine the statistical significance of the difference between two means for the maximum statement nesting depth, we perform the Wilcoxon rank-sum test and obtain the following results:

$$W_1 = 3 + 13 + 13 + 18 + 13 + 7 + 13 + 16 + 17 = 113$$

$$W_2 = 3 + 3 + 3 + 3 + 7 + 9 + 7 + 13 + 10 = 58$$

Thus the null hypothesis $H_0$ for maximum statement nesting depth is rejected at $\alpha =$

| Statement Nesting Levels in Main Programs | | | | | | |
|---|---|---|---|---|---|---|
| Account # | Project 1 | | | Project 2 | | |
| | Executable Statements | Maximum Nesting Depth | Nesting Level per Statement | Executable Statements | Maximum Nesting Depth | Nesting Level per Statement |
| 05 | 8 | 2 | 1.625 | 8 | 2 | 1.625 |
| 06 | 65 | 6 | 3.338 | 18 | 2 | 1.667 |
| 07 | 54 | 6 | 4.333 | 18 | 2 | 1.500 |
| 11 | 33 | 9 | 4.455 | 12 | 2 | 1.500 |
| 12 | 22 | 6 | 3.409 | 14 | 3 | 1.929 |
| 13 | 25 | 3 | 2.560 | 23 | 4 | 2.174 |
| 14 | 34 | 6 | 3.559 | 11 | 3 | 1.909 |
| 19 | 45 | 7 | 4.378 | 17 | 6 | 2.882 |
| 20 | 46 | 8 | 5.000 | 28 | 5 | 2.750 |
| Average | 36.889 | 5.889 | 3.629 | 16.556 | 3.222 | 1.993 |

**Table 8. Statement Nesting Levels in Main Programs**

0.009. Similarly, the rank sums for average nesting depth per statement are calculated as

$$W_1 = 3.5 + 12 + 15 + 17 + 13 + 9 + 14 + 16 + 18 = 117.5$$

$$W_2 = 3.5 + 5 + 1.5 + 1.5 + 7 + 8 + 6 + 11 + 10 = 53.5$$

Again, the null hypothesis $H_0$ is rejected at $\alpha = 0.009$.

## 8.2. Program Comprehension and Modification

Our second study was designed to test how the choice of the different exception handling mechanisms would affect program comprehension and modification. Since this study was conducted after concluding the previous study, the subjects were already familiar with both exception handling mechanisms.

We tested the following pairs of null ($H_0$) and alternative ($H_1$) hypotheses:

$H_0$: the average overall scores for problems in version C is identical to that for problems in version D;
$H_1$: subjects scored higher for problems in version D than for problems in version C.

$H_0$: subjects spent the same amount of time for problems in version C as in version D;
$H_1$: subjects spent more time for problems in version C than version D.

This study took the form of in–class quiz. Each subject was required to solve problems involving a dynamic array (*darray*) and a direct acyclic graph (*dag*). For each problem, the subjects were asked to read a program of three to four pages and then answer several questions, some of which involved comprehension and others modification. In order to investigate the effect of different exception handling mechanisms on program comprehension and modification, we designed two equivalent versions of programs for each of the problems: a control–oriented exception handling version (C) and a data–oriented exception handling version (D). Subjects were assigned to work on one version of the darray problem and another version of the dag problem. The assignments of subjects to versions of the tests were determined randomly. Six subjects worked on version C of darray and version D of dag, and five subjects worked on version D of darray and version C of dag.

Table 9 shows the test scores of the subjects, as well as the time (in minutes) they spent on each problem. The final score assigned to a student on a problem was calculated by counting the number of correct solutions divided by the number of questions.

| Quiz Test Scores and Time Consumptions | | | | | | |
|---|---|---|---|---|---|---|
| | Control–Oriented Method | | | Data–Oriented Method | | |
| Subject | Problem Type | Total Score | Time Used | Problem Type | Total Score | Time Used |
| 1 | Darray | 1.0 | 45 | Dag | 0.67 | 39 |
| 2 | Darray | 0.75 | 50 | Dag | 1.0 | 80 |
| 3 | Darray | 0.5 | 40 | Dag | 0.0 | 86 |
| 4 | Darray | 0.75 | 56 | Dag | 1.0 | 27 |
| 5 | Darray | 0.5 | 61 | Dag | 1.0 | 89 |
| 6 | Darray | 0.75 | 63 | Dag | 0.0 | 27 |
| 7 | Dag | 1.0 | 55.5 | Darray | 1.0 | 55 |
| 8 | Dag | 0.25 | 70.5 | Darray | 1.0 | 63 |
| 9 | Dag | 0.0 | 95 | Darray | 0.5 | 50 |
| 10 | Dag | 0.75 | 53.5 | Darray | 1.0 | 53 |
| 11 | Dag | 0.25 | 40 | Darray | 0.5 | 19 |
| Average | | 0.591 | 57.227 | | 0.697 | 53.455 |

**Table 9. Quiz Test Scores and Time Consumption**

Thus, the final score should always fall into the range of 0.0 to 1.0, inclusively.

The analysis shows that the average score for the problems in version C (0.591) is somewhat lower than the average score for the corresponding problems in version D (0.697). Dividing the questions into two groups (comprehension and modification), we determined that the differences of total scores mainly come from the latter group. Thus, our data-oriented exception handling mechanism may have greater impact on modification activities (a more realistic programming task) than on a programmer's ability to understand and answer questions about a program.

In addition to giving solutions, subjects were also asked to record the time spent on each question. The total time given for the quiz is 150 minutes. The average time spent on a problem for version C was 57.227 minutes, while that for version D was 53.455 minutes. Thus subjects spent less time on problems for data-oriented exception handling versions, but still got better results.

When both of the sample sizes are large (greater than eight), the null distribution of the rank-sum statistic is approximately normal and the test can be performed using the standard normal table. Specifically, the following $Z$ statistic is approximately $N(0,1)$ when $H_0$ is true:

$$Z = \frac{\sum_{i=1}^{m} x_i - \frac{m \times (m + n + 1)}{2}}{\sqrt{\frac{m \times n \times (m + n + 1)}{12}}}$$

where $x_i$ is the $i^{th}$ element of the first sample set. The null hypothesis that both samples come from identical populations can be rejected if the value of a standard variable $Z$ is less than the level of significance of the test. For example, testing the null hypothesis $H_0$ that $\mu_1 = \mu_2$ where $\mu_1$ is the average rank of total scores for problems in version C and $\mu_2$ is its counterpart for problems in version D against the alternative $\mu_1 < \mu_2$, we obtain $Z = 0.886$. By looking up the standard normal statistic table, we find that

$$\mathrm{P}[\ Z < 0.886\ ] = 0.812$$

which is mildly significant. Thus we reject $H_0$ at a level of significance $\alpha = 0.188$. For the hypothesis about the mean amount of time used, the value of the standard variable $Z$ is 0.624. Since

$$\mathrm{P}[\ Z < 0.624\ ] = 0.734$$

we cannot reject $H_0$. The relatively small differences between the respective averages for time consumed and small number of subjects both contribute to the lack of significant differences.

In summary, the studies conducted indicate that data-oriented exception handling can help producing better programs. Since the resulting programs are simpler and better structured, they may be easier to understand and modify.

# CHAPTER 9

# Conclusion

This chapter summarizes the issues discussed in the preceding chapters and identifies some avenues for future investigation.

## 9.1. Summary

Exception handling mechanisms were added to programming languages to separate exception processing from normal cases. Mechanisms were designed to permit invokers to specify responses to exceptions. The result was to be simpler programs and higher quality code.

However, control-oriented exception handling fails to reach the expected goal. Our case study showed that lack of clear definition of exceptions made it difficult to determine what processing was associated with exceptions. Normal cases were often treated as exception processing, e.g., adding a new node to a graph. Raising exceptions such as **`node_not_in_dag`** or **`duplicate_node`** to add a node to a graph only confused the algorithm. Nesting blocks to associate handlers with exceptions interleaved exceptional and normal code, and increased program complexity by increasing statement nesting levels. Exception propagation permits users to specify handler actions, but increases inter-module coupling and the risk of propagating an exception out of its scope. Although control-oriented exception handling was intended to lead to simpler and higher quality code, the case study showed that the resulting code contained unreachable handlers.

As an alternative, we proposed data-oriented exception handling. Exceptions are defined as implementation insufficiencies and are therefore associated with type definitions. Handlers are bound to exceptions in object declarations, centralizing information about handlers and separating exception processing from normal cases. Without exception propagation, our approach still permits users to specify different handler actions for exceptions. On the other hand, the lack of exception propagation in our mechanism makes it less complex than control-oriented exception handling mechanisms.

While the evaluations performed in this research work were not conclusive, all of them pointed in the same direction. The case study showed that even when exceptions were allowed to be raised for conditions other than implementation insufficiencies, (e.g., `makes_cycle`), data-oriented exception handling still increased program quality with reduced statement nesting depth, increased modularity, and centralized and separated handler code. The empirical studies revealed that subjects constructed smaller programs with reduced nesting depth and increased modularity. The program comprehension and modification experiment showed that subjects gained better examination scores in less time. For program testing, we found that while the same amount of test data was needed for programs with different exception handling methods, the version with data-oriented exception handling has fewer exception/handler binding pairs. In addition, the algorithm for monitoring structural coverage test is simpler for data-oriented exception handling than for control-oriented exception handling.

A notable advantage of proof rules for our method is the orthogonality of proofs about exception processing code and proofs of other parts of the program. Adding exceptions to a program requires relatively few changes in an existing proof, all of which occur in declarations. In contrast, in re-establishing the correctness of a program with control-oriented exception handling, many proof steps in different portions of the program need to be done again. Extra specifications for subprogram headings may be required to prove assertions about control-oriented exceptions raised in programs. Additional restrictions are imposed on exception propagation in order to be able to prove programs.

### 9.2. Future Research Directions

It is possible to extend the syntactic and semantic definitions of the primitives in our exception handling mechanism to make it more convenient to use. For example, default handlers for exceptions can be specified in a type declaration and inherited by variables declared with that type.

```
type stack is limited private
     #exception overflow( S : in out stack ) => expand( S, 10 ),
                storage_exhausted( S : in out stack; place : string )
                                               => put_line( place );
```

Thus, the declaration of `S1` and `S2` shown below would cause both data objects to override the default handler for `overflow` (i.e., `expand(S,10)`) and inherit the one for `storage_exhausted`:

```
package integer_stack is new stack_pkg( integer, 20 );
use integer_stack;

S1, S2 : stack
          #when overflow( S : in out stack ) => expand( S, 40 );
```

Subtypes can also be declared in user programs with default handlers. For example:

```
subtype expandable_stack is stack
          #when overflow( S : in out stack ) => expand( S, 40 );
```

Then, the following object declarations

```
S1, S2 : stack
         #when overflow( S : in out stack ) => expand( S, 40 );

S3 : stack
     #when overflow( S : in out stack ) => expand( S, 40 ),
               storage_exhausted( S : in out stack; place : string )
                                         => last_wish( S, place );
```

can be more conveniently declared as:

```
S1, S2 : expandable_stack;

S3 : expandable_stack
     #when storage_exhausted( S : in out stack; place : string )
                                   => last_wish( S, place );
```

More research work is needed to implement our mechanism with such extension, to modify the proof rules discussed in Chapter 6, and to introduce new structural coverage metrics to accommodate the new signaler-handler association patterns.

More experimental studies can also be conducted to compare different exception handling mechanisms. Special tools can be designed to conduct such empirical studies in a more realistic environment. For example, automatic recording of successive program changes during construction or modification of large programs can provide more useful information about how different exception handling mechanisms affect programming practices.

Finally, studies on programming methods, tools and environment associated with the new exception handling mechanism can be conducted. The introduction of better program constructs along with a better programming environment will certainly help increase programming productivity and program quality.

## APPENDIX

# Experimental Studies

## A. Project Assignment : Expandable Hash Table

For this project, you are to design and implement a generic package supporting an abstract data type *hash_table*. A hash table should be represented as a one-dimensional array of variable size, with linear probing to resolve any conflict in hashing. Whenever a key is hashed to an array location already occupied, a linear search through the array is conducted until a free location is found (if the end of the array is reached, the first location of the array is checked in turn.)

The generic package takes the following formal parameters:

```
key_type      -- the type of keys to be hashed
attr_type     -- the type of attributes associated with keys
null_key      -- null value for the key_type
null_attr     -- null value for the attr_type
initial_size  -- initial size for a hash table
maximum_size  -- the maximum allowable size for a hash table
threshold     -- the number of "non-hit" allowed before signaling
hash_func1    -- the original function chosen for hashing
hash_func2    -- the alternate function used for hashing
```

Three exceptions may be raised in the process of hashing:

```
too_many_nonhit  -- when number of "non-hit" exceeds threshold
table_half_full  -- when the hash table is half full
no_more_storage  -- when the table reached maximum_size and is full
```

Each hash table slot holds a pair (`Key, Attr`). All keys in the hash table must be different. If a slot is free, it contains (`null_key, null_attr`). Initially, the hash function `hash_func1` is used to hash keys into a hash table. The hash function associated with a hash table can be switched to `hash_func2` if an operation `switch_hash_function` is invoked. Thus, a simple (though inefficient) function can be chosen as the original hashing function. If necessary, a more efficient (but complicated) function can be used instead. Some research result reveals that the size of a hash table should be twice the cardinality of the set of keys to result in non-linear search time. Therefore, whenever the size of a hash table is half full, an exception is raised. The user of the hash package can then decide whether to invoke an **expand** operation to grant the table more storage. Note that re-hashing may be required after expanding the size of a hash table because the hash function used may be dependent on the size of the hash table. Of course, re-hashing is necessary after hash function switching.

The generic package should provide at least the following operations:

```
create_hash_table      -- create an empty hash table
store_pair             -- store a pair into a hash table
fetch_attr             -- fetch an attribute associated with a key
make_iterator          -- prepare iterating the hash table
more_pair              -- check if there is more pair to iterate
get_next_pair          -- obtain the next pair
switch_hash_function   -- switch from hash_func1 to hash_func2
expand_hash_table      -- expand a hash table by certain amount
change_threshold       -- in order for a user program to proceed after
                          handling the exception too_many_nonhit
```

In addition, you need some inquire functions to test the current state of a hash table.

After implementing the generic package, write a driver procedure to test your package. The driver procedure reads from the input file a sequence of words, and inserts the words into the hash table. For each word, the key to be hashed is the word itself, while its attribute is the word's sequential number. If a word has already been entered into the hash table, just ignore it. After processing all input, or after the hash table reached the maximal allowed size, dump the contents of the hash table.

The actual parameters used to instantiate the generic hash packages are:

```
key_type      => fixed length string with length 20
attr_type     => natural
null_key      => blank string with length 20
null_attr     => 0
initial_size  => 20
maximum_size  => 150
threshold     => 8
hash_func1    => return: (length of the key) mod (current table size)
hash_func2    => return: (sum of ASCII code of characters in the key)
                                 mod (current table size)
```

If `too_many_nonhit` is raised when storing a pair into a hash table, temporarily raise the threshold by one and re-try. However, if the hash table has been expanded 9 times in the past and hash function 1 is currently in use, then switch to hash function 2. Once the pair has been successfully stored into the hash table, the threshold should be restored to its initial value. Note that `too_many_nonhit` can be further raised in the process of function switch. In such a case, the handler raises the threshold by one and then re-tries.

If `table_half_full` is raised, expand the hash table by adding 12 more slots to it. Note that `too_many_nonhit` may be raised when re-hashing. The handle action is similar to that stated in the previous paragraph except that the hash function should be switched after 8 expansions rather than 9. Also, the generic package should not raise **table_half_full** if the maximal size allowed for a hash table has already been reached.

If **no_more_storage** is raised, dump the current contents of the hash table and terminate execution. Also, dump the hash table before switching from hash function 1 to hash function 2.

## B. The Darray Problem ( Control-Oriented Version )

```
generic
    type elem_type is private;        -- Component element type.
package darray_pkg is

    -- This package provides the dynamic array (darray) abstract data type.
    -- A darray has completely dynamic bounds, which change during run-time
    -- as elements are added to/removed from the top/bottom. darrays are
    -- similar to deques, differing only in that operations for indexing
    -- into the structure are also provided. A darray is indexed by
    -- integers that fall within the current bounds.

    type darray is limited private;    -- The darray abstract data type.

    initial_bound : constant := 20;
    maximum_limit : constant := 5 * initial_bound;

    uninitialized_darray : exception;
    out_of_high_bound    : exception;  -- index out of current high bound
    out_of_low_bound     : exception;  -- index out of current low bound
    high_bound_limit_met : exception;  -- maximum high bound limit met
    low_bound_limit_met  : exception;  -- minimum low bound limit met
    storage_exhausted    : exception;  -- exceed pre-declared storage limit

    procedure create( d : in out darray );
    procedure add_high( d : in out darray;  e : elem_type );
    procedure add_low( d : in out darray;  e : elem_type );
    procedure shift_high( d : in out darray; n : positive );
    procedure shift_low( d : in out darray; n : positive );
    procedure expand_high( d : in out darray; amount : positive );
    procedure expand_low( d : in out darray; amount : positive );

private
    ...
end darray_pkg;

---------------------------------------------------------------------------

package body darray_pkg is

    procedure create( d : in out darray ) is
```

```
begin
    d.first_index := 1;
    d.last_index  := 0;
    d.current_high_bound := initial_bound;
    d.current_low_bound  := - initial_bound;
    d.high_bound_limit   := maximum_limit;
    d.low_bound_limit    := - maximum_limit;

    { allocate storage for d }
end create;

procedure add_high( d : in out darray; e : elem_type ) is
begin
    if { d is not initialized } then
        raise uninitialized_darray;
    end if;
    if d.last_index = d.current_high_bound then
        raise out_of_high_bound;
    end if;
    d.last_index := d.last_index + 1;
    { store e into the slot }
end add_high;

procedure add_low( d : in out darray; e : elem_type ) is
begin
    if { d is not initialized } then
        raise uninitialized_darray;
    end if;
    if d.first_index = d.current_low_bound then
        raise out_of_low_bound;
    end if;
    d.first_index := d.first_index - 1;
    { store e into the slot }
end add_low;

procedure shift_high( d : in out darray; n : positive ) is
begin
    if { d is not initialized } then
        raise uninitialized_darray;
    end if;
    if d.last_index + n > d.current_high_bound then
        raise out_of_high_bound;
    end if;
    { shift all elements in d toward the higher end n places }
end shift_high;

procedure shift_low( d : in out darray; n : positive ) is
begin
    if { d is not initialized } then
        raise uninitialized_darray;
    end if;
    if d.first_index - n < d.current_low_bound then
        raise out_of_low_bound;
```

```
        end if;
        { shift all elements in d toward the lower end n places }
    end shift_low;

    procedure expand_high( d : in out darray; amount : positive ) is
    begin
        if { d is not initialized } then
            raise uninitialized_darray;
        end if;
        if d.current_low_bound = d.low_bound_limit and
                    d.current_high_bound = d.high_bound_limit then
            raise storage_exhausted;
        end if;
        if d.current_high_bound = d.high_bound_limit then
            raise high_bound_limit_met;
        end if;
        d.current_high_bound := min( d.current_high_bound + amount,
                                     d.high_bound_limit            );
    end expand_high;

    procedure expand_low( d : in out darray; amount : positive ) is
    begin
        if { d is not initialized } then
            raise uninitialized_darray;
        end if;
        if d.current_low_bound = d.low_bound_limit and
                    d.current_high_bound = d.high_bound_limit then
            raise storage_exhausted;
        end if;
        if d.current_low_bound = d.low_bound_limit then
            raise low_bound_limit_met;
        end if;
        d.current_low_bound := max( d.current_low_bound - amount,
                                    d.low_bound_limit             );
    end expand_low;

end darray_pkg;

----------------------------------------------------------------------------

with darray_pkg, text_io;
use text_io;
procedure main is
    package integer_darray is new darray_pkg( elem_type => integer );
    use integer_darray;

    package int_io is new integer_io( integer );    use int_io;

    d : darray;
    i : integer;
    amount : constant := 20;
begin
    while not end_of_file loop
```

```
        get( i );
loop2:  loop
            begin
                add_high( d, i );
                exit;
            exception
                when uninitialized_darray =>
                    create( d );
                when out_of_high_bound =>
                    begin
                        expand_high( d, amount );
                    exception
                        when high_bound_limit_met =>
                            loop
                                begin
                                    shift_low( d, 10 );
                                    exit;
                                exception
                                    when out_of_low_bound =>
                                        begin
                                            expand_low( d, amount );
                                        exception
                                            when low_bound_limit_met =>
                                                     shift_high( d, 10 );
                                        end;
                                end;
                            end loop;
                    end;
            end;
        end loop;     -- loop2
    end loop;
end main;
```

---

**Question #1 ( Darray )**

(Please record the time you spent on each of the following questions.)

1). Giving an input data file containing enough integers, how many successful `shift_low` operations will be performed? How many `shift_high` operations will be performed?

2). If the input file contains 400 integers, how will the program terminate?

3). Do we need the loop labeled `loop2`? Why?

4). If we change `add_high(d, i)` to `add_low(d, i)`, give the necessary modifications such that the program will perform similarly.

5). Modify the main program such that the following rules are satisfied:

- A new integer is always added to the lower end of the darray;
- If `out_of_low_bound` is raised, shift the contents of the darray toward the higher end over 10 slots;
- If `out_of_high_bound` is raised, expand the high bound by 20 more slots (if possible);
- If `high_bound_limit_met` is raised, first expand the low bound by 20 more slots (if possible), then shift the contents of the darray toward the lower end over 20 slots.

## C. The Darray Problem ( Data–Oriented Version )

```
generic
    type elem_type is private;          -- Component element type.
package darray_pkg is

    -- This package provides the dynamic array (darray) abstract data type.
    -- A darray has completely dynamic bounds, which change during run-time
    -- as elements are added to/removed from the top/bottom. darrays are
    -- similar to deques, differing only in that operations for indexing
    -- into the structure are also provided.  A darray is indexed by
    -- integers that fall within the current bounds.

    type darray is limited private
          #exception uninitialized_darray( d : darray ),
                     out_of_high_bound( d : darray ),
                     out_of_low_bound( d : darray ),
                     high_bound_limit_met( d : darray ),
                     low_bound_limit_met( d : darray )
                     storage_exhausted( d : darray );

    initial_bound : constant := 20;
    maximum_limit : constant := 5 * initial_bound;

    procedure create( d : in out darray );
    procedure add_high( d : in out darray;  e : elem_type );
    procedure add_low( d : in out darray;  e : elem_type );
    procedure shift_high( d : in out darray; n : positive );
    procedure shift_low( d : in out darray; n : positive );
    procedure expand_high( d : in out darray; amount : positive );
    procedure expand_low( d : in out darray; amount : positive );

private
    ...
```

```
end darray_pkg;

------------------------------------------------------------------------

package body darray_pkg is

    ...  -- same as its control-oriented counterpart,
    ...  -- except substituting "#raise" for "raise"

------------------------------------------------------------------------

with darray_pkg, text_io;
use text_io;
procedure main is
    package integer_darray is new darray_pkg( elem_type => integer );
    use integer_darray;

    package int_io is new integer_io( integer );    use int_io;

    amount : constant := 20;
    i : integer;

    d : darray
         #when uninitialized_darray( d : darray ) => create( d ),
               out_of_high_bound( d : darray ) => expand_high( d, amount ),
               out_of_low_bound( d : darray ) => expand_low( d, amount ),
               high_bound_limit_met( d : darray ) => shift_low( d, 10 ),
               low_bound_limit_met( d : darray ) => shift_high( d, 10 );
begin
    while not end_of_file loop
        get( i );
        add_high( d, i );
    end loop;
end main;
```

---

### Question #1 ( Darray )

(Please record the time you spent on each of the following questions.)

1). Giving an input data file containing enough integers, how many successful `shift_low` operations will be performed? How many `shift_high` operations will be performed?

2). If the input file contains 400 integers, how will the program terminate?

3). If we change `add_high(d, i)` to `add_low(d, i)`, give the necessary modifications such that the program will perform similarly.

4). Modify the main program such that the following rules are satisfied:

- A new integer is always added to the lower end of the darray;
- If `out_of_low_bound` is raised, shift the contents of the darray toward the higher end over 10 slots;
- If `out_of_high_bound` is raised, expand the high bound by 20 more slots (if possible);
- If `high_bound_limit_met` is raised, first expand the low bound by 20 more slots (if possible), then shift the contents of the darray toward the lower end over 20 slots.

## D. The Dag Problem ( Control–Oriented Version )

```
generic
    type label is private;      -- labels of nodes
package generic_dag_pkg is

    type dag is private;        -- the dag abstract data type.

    uninitialized_dag : exception;
    node_not_in_dag   : exception;
    duplicate_node    : exception;
    duplicate_edge    : exception;
    makes_cycle       : exception;

    function  create return dag;
    procedure add_node( g : in out dag; l : label );
    procedure add_edge( g : in out dag; l1, l2 : label );
    procedure check_node( g : dag; l : label );

private
    ...
end generic_dag_pkg;
---------------------------------------------------------------------------

package body generic_dag_pkg is

    function create return dag is  { ... }  end create;

    procedure add_node( g : in out dag; l : label ) is
    begin
        if { g has not been initialized } then
            raise uninitialized_dag;
        end if;
```

```
        if { there is already a node in g with label l } then
            raise duplicate_node;
        else
            { add a node with label l to g }
        end if;
    end add_node;

    procedure add_edge( g : in out dag; l1, l2 : label ) is
    begin
        if { g has not been initialized } then
            raise uninitialized_dag;
        end if;
        if { there is no node in g with label l1 } then
            raise node_not_in_dag;
        end if;
        if { there is no node in g with label l2 } then
            raise node_not_in_dag;
        end if;
        if { an edge from node labeled l1 to node labeled l2 is in g } then
            raise duplicate_edge;
        elsif { the new edge will introduce a cycle in g } then
            raise makes_cycle;
        else
            { add the new edge to g }
        end if;
    end add_edge;

    procedure check_node( g : dag; l : label ) is
    begin
        if { g has not been initialized } then
            raise uninitialized_dag;
        end if;
        if { there is no node with label l in dag g } then
            raise node_not_in_dag;
        end if;
    end check_node;

end generic_dag_pkg;

----------------------------------------------------------------------------

    with generic_dag_pkg;
    procedure build_dag is
        package units_dag_pkg is new generic_dag_pkg( character );
        use units_dag_pkg;

        withs_dag, cycle_dag : dag;

        procedure put_info_in_dag( parent_node: character;
                                   withs_list : string ) is separate;
    begin
        withs_dag := create;
        cycle_dag := create;
```

```
3:      put_info_in_dag( 'B', "A" );     -- with A;    package B is ...
4:      put_info_in_dag( 'C', "AB" );    -- with A, B; package C is ...
5:      put_info_in_dag( 'A', "C" );     -- with C;    package A is ...
    end build_dag;
```

---

```
    separate( build_dag )
    procedure put_info_in_dag( parent_node : character;
                                 withs_list : string ) is
        with_node : character;

        procedure add_to_cycle_dag( parent_node,
                                    with_node : character ) is separate;
    begin
        begin
6:          check_node( withs_dag, parent_node );
        exception
            when node_not_in_dag =>
7:               add_node( withs_dag, parent_node );
        end;

8:      for I in withs_list'range loop
9:          with_node := withs_list(I);
            begin
10:             add_node( withs_dag, with_node);
11:             add_edge( withs_dag, parent_node, with_node );
            exception
                when duplicate_node =>
                    begin
12:                     add_edge( withs_dag, parent_node, with_node );
                    exception
                        when makes_cycle =>
                            begin
13:                             add_to_cycle_dag( parent_node, with_node);
                            exception
                                when duplicate_node | makes_cycle =>
14:                                     null;
                            end;
                        when duplicate_edge =>
15:                             null;
                    end;
                when makes_cycle =>
                    begin
16:                     add_to_cycle_dag( parent_node, with_node);
                    exception
                        when duplicate_node | makes_cycle =>
17:                             null;
                    end;
                when duplicate_edge =>
18:                     null;
            end;
        end loop;
```

```
    end put_info_in_dag;
```

---

```
    separate( build_dag.put_info_in_dag )
    procedure add_to_cycle_dag( parent_node, with_node : character ) is
    begin
19:     add_node( cycle_dag, parent_node);
20:     add_node( cycle_dag, with_node);
21:     add_edge( cycle_dag, parent_node, with_node);
    end add_to_cycle_dag;
```

---

## Question #2 ( Dag )

(Please record the time you spent on each of the following questions.)

1). Execute the program "by hand", complete the following trace table:

| Statement # | exception raised | withs_dag | cycle_dag |
|---|---|---|---|
| 3 | | node = { }<br>edge = { } | node = { }<br>edge = { } |
| 6 | node_not_in_dag | ditto | ditto |
| 7 | | node = { 'B' }<br>edge = { } | ditto |
| 10 | | node = { 'B', 'A' }<br>edge = { } | ditto |
| 11 | | node = { 'B', 'A' }<br>edge = { ('A', 'B') } | ditto |
| | | | |
| | | | |
| | | | |

2). Is statement 13 reachable? If the answer is "yes", what is the minimum input data for control to reach it? If the answer is "no", give your justification.

3). Re-do question 2) for statement 16.

4). Suppose that we forgot to initialize the two dags in **build_dag** (i.e., statements 1 and 2 were missing in the program). Modify the program by adding some exception handlers for **uninitialized_dag** such that the program will perform correctly.

## E. The Dag Problem ( Data-Oriented Version )

```
generic
    type label is private;      -- labels of nodes
package generic_dag_pkg is

    type dag is private
            #exception uninitialized_dag( g : dag ),
                       duplicate_node( g : dag; l : label ),
                       node_not_in_dag( g : dag; l : label ),
                       duplicate_edge( g : dag; l1 : label; l2 : label ),
                       makes_cycle( g : dag; l1 : label; l2 : label);

    function  create return dag;
    procedure add_node( g : in out dag; l : label );
    procedure add_edge( g : in out dag; l1, l2 : label );
    procedure check_node( g : dag; l : label );

private
    ...
end generic_dag_pkg;

---------------------------------------------------------------------------

package body generic_dag_pkg is

    ...  -- same as its control-oriented counterpart,
    ...  -- except substituting "#raise" for "raise"

end generic_dag_pkg;

---------------------------------------------------------------------------

    with generic_dag_pkg;
    procedure build_dag is
        package units_dag_pkg is new generic_dag_pkg( character );
        use units_dag_pkg;

        withs_dag : dag;

        cycle_dag : dag
                  #when duplicate_node(g: dag; l: label) =>
1:                                                              null,
                        makes_cycle(g: dag; l1: label; l2: label) =>
2:                                                                   null;
        procedure put_info_in_dag( parent_node: character;
                                   withs_list : string ) is separate;
    begin
3:      withs_dag := create;
4:      cycle_dag := create;
5:      put_info_in_dag( 'B', "A" );    -- with A;    package B is ...
6:      put_info_in_dag( 'C', "AB" );   -- with A, B; package C is ...
```

```
7:      put_info_in_dag( 'A', "C" );     -- with C;    package A is ...
    end build_dag;
```

---

```
    separate( build_dag )
    procedure put_info_in_dag( parent_node : character;
                                     withs_list : string ) is
        with_node : character;

        procedure add_to_cycle_dag( parent_node, with_node : character )
                                                           is separate;
        temp_dag : dag
                 #when node_not_in_dag(g: dag; l: label) =>
8:                                                 add_node( g, parent_node ),
                       duplicate_node(g: dag; l: label) =>
9:                                                          null,
                       duplicate_edge(g: dag; l1: label; l2: label) =>
10:                                                                 null,
                       makes_cycle(g: dag; l1: label; l2: label) =>
11:                                               add_to_cycle_dag(l1, l2 );
    begin
12:     temp_dag := withs_dag;
13:     check_node( temp_dag, parent_node );
14:     for I in withs_list'range loop
15:         with_node := withs_list(I);
16:         add_node( temp_dag, with_node);
17:         add_edge( temp_dag, parent_node, with_node );
        end loop;
18:     withs_dag := temp_dag;
    end put_info_in_dag;
```

---

```
    separate( build_dag.put_info_in_dag )
    procedure add_to_cycle_dag( parent_node, with_node : character ) is
    begin
19:     add_node( cycle_dag, parent_node);
20:     add_node( cycle_dag, with_node);
21:     add_edge( cycle_dag, parent_node, with_node);
    end add_to_cycle_dag;
```

---

## Question #2 ( Dag )

(Please record the time you spent on each of the following questions.)

1). Execute the program "by hand", complete the following trace table:

| Statement # | exception raised | withs_dag / temp_dag | cycle_dag |
|---|---|---|---|
| 5 | | node = { }<br>edge = { } | node = { }<br>edge = { } |
| 13 | node_not_in_dag | ditto | ditto |
| 8 | | node = { 'B' }<br>edge = { } | ditto |
| 14 | | ditto | ditto |
| 16 | | node = { 'B', 'A' }<br>edge = { } | ditto |
| 17 | | node = { 'B', 'A' }<br>edge = { ('A', 'B') } | ditto |
| | | | |
| | | | |
| | | | |

2). What is the minimum input data for control to reach statement 11?

3). Suppose that we forgot to initialize the two dags in `build_dag` (i.e., statements 3 and 4 were missing in the program). Modify the program by adding some exception handlers for `uninitialized_dag` such that the program will perform correctly.

## REFERENCES


[Ada 82]
US Department of Defense, *Reference Manual for the Ada Programming Language,* AdaTEC (July 1982).

[Bentley 87]
J. Bentley and D. Gries, *Programming Pearls,* CACM Vol. 30, No. 4, pp. 284–289 (April 1987).

[Bhatt 77]
G. K. Bhattacharyya and Richard A. Johnson, *Statistical Concepts and Methods,* published by John Wiley & Sons, Inc., (Aug. 1977).

[Black 83]
A. P. Black, *Exception Handling: the Case Against,* Ph.D. Dissertation, TR 82-01-02, Dept. of Computer Science, Univ. of Washington (May 1983).

[Cristian 84]
F. Cristian, *Correct and Robust Programs,* IEEE Trans. Software Eng., Vol. SE-10, No. 2, pp. 163–174 (March 1984).

[Dony 89]
C. Dony, *Object-oriented Design Improves Exception Handling,* European Conference on Software Eng., Toulouse, France (Dec. 1989) [*to appear*].

[Floyd 67]
R. W. Floyd, *Assigning Meanings to Programs,* In J. T. Schwartz (editor), *Proceedings of a Symposium in Applied Mathematics,* Vol. 19, pp. 10–32, American Mathematical Society (1967).

[Goodenough 75]
J. B. Goodenough, *Exception Handling: Issues and a Proposed Notation,* Commun. ACM Vol. 18, No. 12, pp. 683–696 (Dec. 1975).

[Hoare 69]
C.A.R. Hoare, *An Axiomatic Basis for Computer Programming,* CACM Vol. 12, (Oct. 1969).

[Horning 79]
J. J. Horning, *Effects of Programming Languages on Reliability,* in *Computing System Reliability,* (T. Adenson and B. Randell, eds.), Cambridge University Press, (1979).

[Ichbiah 79]
J. D. Ichbiah, et al., *Rationale for the Design of the Ada Programming Language,*

SIGPLAN Notices, Vol. 14, No. 6, Part B (June 1979).

[Knudsen 87]
J. L. Knudsen, *Better Exception Handling in Block Structured Systems,* IEEE Software, pp. 40–49, (May 1987).

[Levin 77]
R. Levin, *Program Structures for Exceptional Condition Handling,* Ph. D. Dissertation, Carnegie-Mellon University (June 1977).

[Liskov 79]
B. H. Liskov and A. Snyder, *Exception Handling in CLU,* IEEE Trans. Software Eng., Vol. SE–5, No. 6 (Nov 1979).

[Luckham 80]
D. C. Luckham and W. Polak, *Ada Exception Handling: An Axiomatic Approach,* ACM Trans. Prog. Lang. and Syst. Vol. 2, No. 2, pp. 225–233 (April 1980).

[MacLaren 77]
M. D. MacLaren, *Exception Handling in PL/I,* SIGPLAN Notices, Vol. 12, No. 3, pp. 101–104 (March 1977).

[McGettrick 82]
A. D. McGettrick, *Program Verification using Ada,* Cambridge University Press, (1982).

[Mitchell 79]
J. G. Mitchell, W. Maybury, and R. Sweet, *Mesa Language Manual Version 5.0,* CSL 79–3, Xerox Palo Alto Research Center (April 1979).

[Parnas 76]
D. L. Parnas and H. Wurges, *Response to Undesireable Events in Software Systems,* Proceedings 2nd International Conference on Software Engineering, pp. 437–446, (1976).

[PL/I 76]
American National Standards Institute, *Programming Language PL/I,* X3.53 (1976).

[Wulf 76]
W. A. Wulf, R. L. London, and M. Shaw, *An Introduction to the Construction and Verification of Alphard Programs,* IEEE Trans. Software Eng., Vol. SE–2, No. 4, pp. 253–265 (Dec. 1976).

[Yemini 85]
S. Yemini and D. M. Berry, *A Modular Verifiable Exception-Handling Mechanism,* ACM Trans. Prog. Lang. and Syst., Vol. 7, No. 2, pp. 214–243 (April 1985).