ANOMALIES ENCOUNTERED IN ADA EXCEPTION HANDLING

C. A. Koeritz
J.C. Knight

Computer Science Report No. TR-93-11
January 26, 1993

Anomalies Encountered in Ada Exception Handling[†]

C. A. Koeritz
J. C. Knight

Department of Computer Science
Thornton Hall
University of Virginia
Charlottesville, VA 22903

Abstract

The correct handling of exceptions is important in dependable computing systems. The exception semantics in the Ada language provide a means for handling exceptions, but the mere semantics do not guarantee any properties of dependability by themselves. A set of unusual situations called anomalies are identified in Ada exception handling. The presence of an anomaly in an Ada program can cause the program to diverge from its specification, often in a completely unacceptable manner. Anomalies are introduced into Ada programs through inadequate implementations of the exception handling portions of the program and are not due to normal program execution. Because anomalies exist in the exception handling portions of Ada programs, it is difficult to detect the presence of anomalies during implementation, it is hard to handle their consequences during execution, and they are capable of causing unexpected results in systems that are intended to be dependable. In this paper, the known anomalies in Ada exception handling are enumerated and each is analyzed in terms of its causes and consequences.

[†]This work sponsored in part by the MITRE Corporation and in part by the Virginia CIT under grant number INF-92-002.

Acknowledgements

Much of the technical material described in this document is either the work of our colleagues at MITRE, Chuck Howell, Diane Mularz and Gary Bundy, or work performed jointly with them. This report is a compendium documenting these efforts. This work is sponsored in part by the MITRE Corporation and in part by the Virginia CIT under grant number INF-92-002.

Table of Contents

Figures

Index of Definitions

1. Introduction

This research is addressing problems that have been encountered in the Ada language's exception handling semantics. Exception handling is intended to enhance program dependability by providing a language-defined method for signalling that an invoked subprogram cannot fulfill the primary service requested from it. A single language mechanism is provided for signalling exceptional conditions instead of each program using its own ad hoc condition signalling.

The problems addressed here are collectively called *anomalies*, because their presence can cause a program to exhibit unexpected and unusual behavior when an exception is raised. For example, an unexpected result of one anomaly is that the program may complete before fulfilling its purpose. Another unusual possibility is that the program may deadlock—activity within it ceases as two units wait for each other forever, neither making forward progress. If anomalies such as these exist in an application, then certain properties of dependability can of course not be guaranteed.

This paper introduces the known anomalies. The relevant terms are presented in the first section. The next four sections present groups of related anomalies. Each anomaly is described informally and then discussed in terms of the semantics of Ada that permit its existence. Undesirable consequences are presented for each anomaly, and a formal English definition of each anomaly is given.

2. Terminology

The technical terms used in this paper are drawn primarily from the Ada Language Reference Manual [1]. It is assumed that the reader is familiar with the basics of programming languages and is relatively familiar with programming in Ada. The Ada semantics describing exception handling will be reviewed here to provide an introduction to the topic, and a few additional terms will be introduced also.

The Ada language provides both *procedures* and *functions*, facilitating the decomposition of a programming goal into many smaller sub-goals. The term *subprogram* is used to refer to either a procedure or a function. Subprograms are composed of block statements. A block is usually a small piece of algorithm that has its own declaration section.

Groups of related procedures and functions, along with their data types, are collected into *packages*. A package usually implements a single abstract data type by providing initialization and manipulation routines that operate on the type. *Generic packages* operate on types that are specified after the package is implemented. Genericity allows abstractions to be constructed orthogonally to the type of data contained within the abstraction.

The processing of information in parallel is supported by the *task* in Ada. Communication with tasks is accomplished through rendezvous by making a task *entry* call. To respond to requests for service, the task executes an *accept* statement. Each task executes independently from the other tasks in the system until a rendezvous is initiated.

Exceptions are used in Ada to allow partial algorithms to be made more robust. A partial algorithm is valid on some restricted set of the total domain it might be applied to, while a robust algorithm is defined over its entire domain. When a subprogram has exceptions incorporated into its specification, some section of the domain is represented by raising an exception. The exception covers a particular hole in the domain and extends the region of applicability of the subprogram.

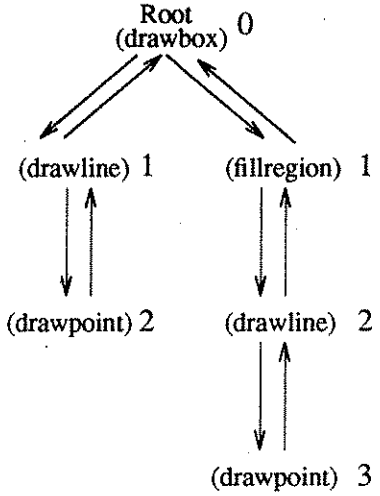
To signify that a subprogram has been invoked in a section of its exceptional domain, an exception is *raised. Handling* of an exception occurs when the exception handlers attached to the bottom of a frame are activated by a raised exception. An exception handler may either *mask* the exception, causing a normal completion of its subprogram, or it may *propagate* the exception, causing an exceptional completion of the subprogram. Exceptions that are propagated are handled by the invoker of the frame, and exception handling continues until the exception is masked or the program is completed.

In Ada, exceptions have static visibility rules that are similar to the visibility rules for variables. An exception can be raised or handled by its own name in locations where it is visible. This region of visibility is called the exception's *scope*. All frames to which an exception's declaration is visible are included within the exception's scope.

The smallest program structure within which an exception can be declared, raised or handled is called a *frame*. Exceptions are propagated between frames when a frame propagates an exception to its immediate invoker. A frame can be any one of the Ada structures that has its own declarative section, including block statements, procedures, functions, task and package bodies, and generic package bodies. When exceptions are declared in the specifications for packages, tasks and generic packages, their scope extends to all frames with visibility to the specification.

A term related to the frame is the *operation*, which connotes a frame that also has an interface specification of some sort. Block statements would not normally be considered operations unless a specification for their operation is provided. Operations can be discussed in terms of their goals as well as their implementations.

A term that is used extensively is the *call depth* of an operation. The call depth is a dynamic value; it depends upon the order in which frames are invoked during the execution of an Ada program. The root frame has a call depth of 0. The call depth measures the level of nesting in that occurs due to subprogram invocations, including the nesting of invoked block statements within a subprogram. To illustrate call depth, Figure 1 contains a *call graph* for a small Ada program. The graph is a tree representation of the path of control flow in the program fragment. In the figure, the call depth increases when a directed edge connects one location to another location that is farther from the root frame. For example, the call depth increases when the arc from the `drawbox`




```
procedure drawbox
   (left_top, right_bottom : in points) is
begin
   drawline(left_top.x, left_top.y, left_top.x, right_bottom.y);
    ... other sides of box ...
   fill_region(left_top, right_bottom);
end drawbox;

-- fillregion repetitively calls drawline

-- drawline repetitively calls drawpoint
```

Figure 1: Sample Call Graph

procedure is followed to the `drawline` procedure.

The call depth decreases every time a subprogram or block statement completes. Call depth can be used to analyze the flow of control during by exception propagation. Propagation is represented by a decrease in call depth, because a frame is completed after it propagates an exception.

In the sequel, definitions for each anomaly will be presented. Each of these definitions is a formalized English statement describing the circumstances in which the anomaly occurs, and the terms of the definition are chosen mainly from the terms presented in this section. It may be convenient to refer back to this section in order to better understand the anomaly definitions.

3. Propagation Related Anomalies

The propagation mechanism in Ada is flawed. There are a number of points that support this claim:

(1) Exceptions are allowed to propagate when they have not been explicitly declared as propagatable, and indeed it is impossible to declare which exceptions are propagatable in standard Ada.

(2) The five predefined exceptions are raised by numerous different operations. There is no standard definition for which of these exceptions are raised by the full set of Ada operations. Thus there is no definite description for how particular Ada operations will behave. This situation can result in an unexpected exception being raised by a subprogram due to one of these five exceptions.

(3) Propagation is a dynamic activity that depends on the order of operation invocation. Exceptional scope is static and depends on program structure. The differences between these two are difficult to reconcile.

(4) Tasking and propagation are not integrated together well with respect to managing the responsibility for restarting tasks that have completed prematurely.

3.1. Uncontrolled Propagation

Exceptions are propagated out of an Ada operation when they are not masked by it. Propagation is relied upon to alert an operation's invoker about a condition arising in the operation that it cannot handle internally. Control of propagation is lost when an exception is not handled in the appropriate frame and is allowed to propagate. Since the exception propagates out of the frame where it should have been handled, it is likely that the invoker's frame will not contain an appropriate exception handler for that exception. Depending on the number of times the exception propagates, it "unwinds" the program accordingly and the likelihood of properly handling the exception decreases.

Ada does provide the *when others* exception handler to handle any exceptions that are not explicitly handled by other exception handlers. This allows a *firewall* to be constructed in a program. This stops exceptions from propagating past the operation containing the firewall. The *when others* handler still causes completion of the operation containing it however, but this can be a normal completion rather than an exceptional completion.

Programs with firewalls are easier to verify than programs that allow uncontrolled propagation, but only for properties of continued execution in the face of unhandled exceptions. There is no logic in the *when others* handler associated with stating why it was invoked for a particular exception (its pre-conditions) or what the handler's goals are for that exception (its post-conditions). It is therefore difficult to prove useful properties for programs containing firewalls compared to programs that handle each exception in a locale where that exception has meaning.

Definition 1: Uncontrolled Propagation Anomaly ::= Consider a subprogram frame called F_n that has been invoked and resides at call depth n. If an exception named E is raised in frame F_n, and F_n either: 1) does not have a handler for exception E or 2) F_n's handler for E can reraise E, then E can propagate to depth $n-1$. The frame $F_{(n-1)}$ may propagate E again, and the propagation might continue until the root program completes. We define uncontrolled propagation to be either: 1) an unexpected

completion of the root program due to exception E's propagation, or 2) a propagation of E greater than $N_{Threshold}$ times, where $N_{Threshold}$ is a maximum number of propagations allowed for a particular application.

3.2. Unnoticed Task Completion

Ada tasks execute partially independently from the operation that initiates them—the operation communicates with the task by rendezvousing with it, but the task begins the rendezvous according to its own implementation rather than being bound by the operation's request for rendezvous.

When a task is executing and an exception is raised, handling proceeds as if the task were its own root program. If the exception propagates to the outermost frame of the task, then the task completes. When this occurs, the task no longer exists to service the requests for which it was designed. Ada deals with this situation by raising the exception named TASKING_ERROR when an operation attempts rendezvous with the task. Notice however that TASKING_ERROR is not raised until that rendezvous is attempted, meaning that the first indication of the task's completion to the rest of the program occurs at a time when the task's services are needed. This delay between the completion of the task and the notification of the task's completion could be used profitably to restart the task, but it is not.

Further, it may be that the operation attempting rendezvous is not intended to restart the task. The TASKING_ERROR exception may have to be propagated to an exception handler that is capable of restarting the task.

A more pernicious effect of the unnoticed task completion is that there is no record of *why* the task completed. While ad hoc techniques such as message passing or writing to shared variables can record information about the task's demise, that information may not even be recorded when the task completes. The fact that a task completes due to a raised exception indicates either that the outermost frame did not handle the exception, or that it handled and then masked the exception. In the first case, the lack of a handler means that the task was given no opportunity to record the exception at the outermost level. Only by recording the exception raise before the outermost frame can the reason for that task's completion be known by the rest of the program.

The second case, in which the task handles the exception at the outermost level and then completes, allows the task to record the exception raise as long as each handler at that level performs the recording operation. In either case, however, several steps must be taken to ensure that the reason for the task's completion is recorded—Ada does not maintain the information.

Definition 2: Unnoticed Task Completion Anomaly ::= Consider an exception *E* that is raised in a task *T*. If task *T* completes unexpectedly due to exception *E*, then the exception `TASKING_ERROR` is raised in any frame *F* that attempts to rendezvous with task *T*. The exception *E* is not raised at the point of rendezvous with the task, although the raise of *E* is the actual reason for task completion.

3.3. Anonymous Exceptions

Exceptions have a *scope* in which they are visible, and correspondingly have regions where they are not visible. The scope of an exception is defined statically by the program declaring it; it consists of all operations to which the exception's declaration is visible. An anonymous exception is an exception whose name has "disappeared" because the exception propagates out of scope.

Consider the program fragment in Figure 2. The outer frame **A** contains several program statements (not shown), including the inner frame **B**. Frame B declares an exception called `Event_Error`. At location (1) in frame B, `Event_Error` is raised. Since block B has no handler for `Event_Error`, B is completed and the exception propagates to frame A. The propagation causes the `Event_Error` to be raised at location (2) in frame A.

After this sequence of events, the exception `Event_Error` is anonymous. It is declared in frame B and is meaningless within frame A. Frame A handles the exception at location (3) with the `others` exception handler. This is in fact the only way an anonymous exception *can* be handled. Anonymous exceptions are dangerous to programs, because any logic that the programmer encodes specifically for that exception is useless when the exception becomes anonymous.

Definition 3: Anonymous Exception Anomaly ::= An exception *E* is declared by operation *O*. Exception *E*'s scope *S* is the set of all operations that have visibility of *O*. If exception *E* propagates out of *S*, then *E* becomes an anonymous exception. The anonymous exception never reacquires its previous name of *E*, even if it is propagated back into an operation residing in its scope *S*.

3.4. Anonymous Exceptions from Generic Elaborations

When a generic package is instantiated with a particular type, there must be an elaboration of the package before it can be used. Elaboration creates a new unit to work on the specified type. During elaboration the variables declared in the generic package are initialized and an exception might be raised during the initialization (for example, if a variable is initialized to the value returned by a function, and the function raises an exception).

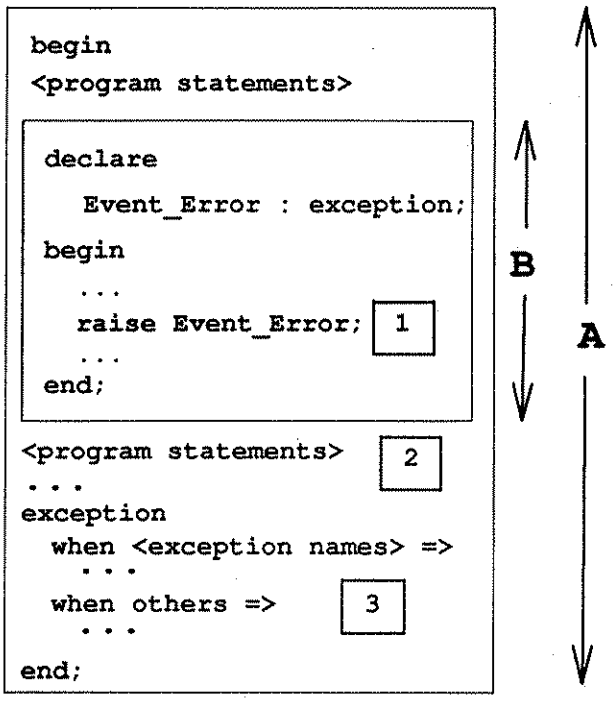


Figure 2: Anonymous Exception Example

The elaboration of generic packages in Ada can therefore raise an exception, indicating that the generic package was not successfully instantiated. While the possibility of an aborted elaboration is detailed in section 11.4.2 of the Ada LRM [1], the fact that an aborted elaboration can raise an anonymous exception is not reported there [2]. The reason an anonymous exception might be raised is due to an interaction between the scope of an exception and the scope of the generic package.

The generic package would normally be included in the scope of the exception being raised, since the generic package is importing the operation that raised the exception. However, the generic package does not actually exist until after it has been elaborated. The frame attempting to elaborate the generic package is where the exception is actually raised, because there are as yet no valid frames within the generic package. While the operations inside the generic package would have visibility to the exception after elaboration, the operations outside of the package are not necessarily in the exception's scope. And of course, an exception that is propagated out of its scope becomes anonymous.

This is a pernicious problem because the designer of the generic package may have forgotten that elaboration occurs in a different scope than the generic package. The

exception may be raised only under peculiar circumstances that evade testing, and then the clients of the generic package will have to contend with an unanticipated anonymous exception when the package is elaborated.

Definition 4: Anonymous due to Elaboration Anomaly ::= Consider a generic package *G* with a group of variables that are initialized when *G* is elaborated. Frame *F* attempts to elaborate *G*. If an exception is raised during *G*'s elaboration, it is anonymous if frame *F* does not have visibility to *E*. Since *F* relies upon package *G* to implement the generic abstraction, it is unlikely that an internal exception will be visible to *F*. Even if the generic package *G* has handlers for exception *E* in its package body, the handlers are not activated because *G* has not completed elaboration.

3.5. New Exceptions Raised During Handling

In Ada, many operations, even basic mathematical operations such as addition, can raise an exception under the right set of circumstances. These same operations are used to implement exception handlers. Thus, exception handlers can potentially raise exceptions in ways other than the explicit *raise* statement. If an exception is raised within an exception handler's outermost level (outside of any nested blocks that may have their own exception handlers), then the frame containing that exception handler is completed and the exception is propagated to the invoker of the frame. Note that this exception is probably different than the exception that the handler was intended to handle—that exception is completely forgotten about. Even if the handler had only partially completed the handling of its exception, the new exception is the only one that the invoker is informed of.

Not only is the old exception completely forgotten about, but the invoker might assume that the new exception was raised by the frame's body and not by the frame's exception handlers. In this case, the exception handling that the invoker performs may not be what would otherwise be intended for the new exception.

Definition 5: Forgotten Exception Anomaly ::= Consider an exception *E* raised in a frame *F*. If a handler *H* for *E* is partially complete when an exception E_2 is raised at the outermost level in *H*, then exception *E* is lowered (the opposite of raised). Exception *E* is no longer handled by the frame. Exception E_2 is propagated and completes frame *F* instead of exception *E*.

3.6. Deadlock due to Propagation Delay

Tasks are dependent upon the operation that instantiates them. If the capability of communicating with a task is lost, then it is useless to the Ada program because no results will ever be received from it and no control commands can affect it. The control of communication with the task resides in the instantiating operation, and thus tasks must

complete when the operation upon which they are dependent completes. To do otherwise would sever the task from the rest of the program and it would be useless at that point. Both normal and exceptional completions of the instantiating operation require any dependent tasks to be completed.

Section 9.4.6 of the Ada reference manual [1] states that if a frame has a dependent task and an exception is raised in the frame, then the propagation of that exception is delayed until all of the dependent tasks complete. Either the tasks must complete of their own volition or be explicitly aborted. The raised exception is handled by the frame's exception handlers, but after execution of the handler the frame is not allowed to complete until its dependent tasks also complete execution.

This delay of propagation can introduce deadlock into Ada programs. An example of this is shown in Figure 3. The procedure `DeadLocker` has a dependent task that blocks awaiting a rendezvous. If an exception is raised before that rendezvous, the task will never begin execution after the accept statements—the task is waiting for the procedure to rendezvous. A deadlock results because the procedure must wait for the completion of its dependent task before its own completion, and that completion will never occur because the task is waiting for the procedure as well.

Definition 6: Delayed Propagation Anomaly ::= A frame *F* instantiates a set of tasks *T*. An exception *E* is raised in frame *F* after the tasks are instantiated. If *E* has an exception handler attached to frame *F*, then *E* is handled, but *E* cannot propagate until all dependent tasks in *T* are completed.

3.7. Termination of Servers

Tasks are used in some applications to provide a service with an indefinitely long or effectively infinite lifetime. It might be that the task provides a service for the entire duration of the application program's execution. If an exception is raised in such a task and the task completes, then the service is no longer available to the system. If the service was of critical importance to the continued operation of the program, then not only has the specification for the service been violated, but the system specification might be violated as well.

Definition 7: Persistent Server Anomaly ::= A task *T* is specified as providing a service that is required throughout the duration of program execution. If an exception *E* is raised in task *T*, and task *T* completes, then task *T*'s specification of persistence is violated.

```
with Text_Io; use Text_Io;
with Calendar; use Calendar;

procedure Deadlocker is
  package Day_Duration is new Fixed_Io (Day_Duration);
  use Day_Duration_Io;

  Death_exception : exception;

begin
  Example_Block:
    declare
      task Sleeper is
        entry Go_Ahead;
      end;
      task body Sleeper is
        begin
         accept Go_Ahead;
          delay 10.0;
          Put("Seconds on clock after sleep: ");
          Put(Seconds(Clock));
          New_Line;
        end Sleeper;

    begin  -- of Example_Block
      if Seconds(Clock) > 0.0 then
        raise Death_exception;
      end if;

    Sleeper.Go_Ahead;  -- never reached

    exception
      when Death_exception =>
        Put("Exception handled in block at ");
        Put(Seconds(Clock));
        New_Line;
        raise;
    end Example_block;

exception
  when Death_exception =>
    Put("Exception handled in main procedure at ");
    Put(Seconds(Clock));
    New_Line;

end Deadlocker;
```

Figure 3: Example of Deadlock due to Propagation Delay

4. Anomalies Related to Specification Breakdown

The anomalies in this section result when the specification for an operation is not honored. Generally, a specification consists of pre-conditions and post-conditions for the operation. The pre-conditions details the conditions under which the operation can be invoked. If the pre-conditions are not satisfied, then the behavior of the operation is not defined. Post-conditions specify what the operation is to achieve. They are supposed to be guaranteed at the time the operation completes.

Operations should be designed according to their specification, but are not necessarily designed that way. In a perfect world, operations could assume that their pre-conditions are honored and that they are not invoked in a state that violates the pre-conditions, but in reality this is not the case. If an operation does not check its pre-conditions, it may do worse than raise an unexpected exception: the program state could be made inconsistent by the incorrect assumption.

Conversely, an operation should always be implemented to guarantee that the post-conditions are met. But in cases where an exception is raised during execution of the operation, it may be impossible to fulfill the post-conditions completely. Often a weaker form of the post-conditions are guaranteed for particular exceptions, while for other exceptions the post-conditions cannot be met at all.

Anomalies arise when either the pre-conditions or post-conditions of the specification are violated. Some aspects of Ada seem to lend themselves to making it difficult to meet an operation's specification. These aspects are the subject of this section.

4.1. Exceptions and Abstractions

Ada is object based, meaning that abstract data types may be constructed and packages may be defined to operate on the abstract types. Larger abstractions may be derived from smaller ones by importing pre-existing abstractions. This technique of building up large objects out of smaller and potentially reusable components is a powerful tool in the construction of programs because the derived abstraction can present an utterly different interface to its clients. The clients make use of a derived type without needing to understand the details of the base types that it uses. However, exception handling can cause a break down in the interface to the derived abstraction because an exception specific to a particular low-level data type can be propagated to a higher-level package making use of that type.

An exception may be raised in a subprogram that operates on the base type of the abstraction [2]. This exception may be visible to the client of the higher-level abstraction, but probably will not be. The client expects that the abstraction's interface is accurate and specifies any exceptions that can be propagated from the operations on the higher-level abstraction. Since the derived type's operations did not handle one of the base type's exceptions, the exception is anonymous unless the client also imports the base type's interface. This is a violation of the intentions of the abstraction designer, since a lower-level type must be imported to effectively use a type based on it.

The important point here is not that the derived type omitted an exception handler for one of the base type's exceptions. The point is that Ada does not automatically make the base type's exceptions visible to the clients of the derived type. A more reasonable approach would extend the scope of the exception to all operations making use of types containing the exception, rather than forcing the designer of the client operation to

include an importation of the base type. With proper documentation in the derived type's package specification, the visibility of the base type's exception would allow handling without extra importations. In the current definition of Ada, the base type's package must be imported or exceptions raised by it in the clients of the derived type will be anonymous.

Definition 8: Abstraction Breakdown Anomaly ::= A package implementing an abstraction A_2 uses an abstraction A_1. A client of the more complex abstraction A_2 should not be forced to deal with implementation details of the base abstraction A_1, but exceptions raised in A_1 might be propagated through A_2 to the client. An exception E propagated in that manner is a violation of the abstraction A_2, because the specification of A_1 must be examined to correctly use A_2. Further, E's scope may not include the client of A_2, because A_2 is the direct user of abstraction A_1, in which case exception E becomes anonymous in the client. The exception E is not "inherited" by the derived abstraction A_2 and cannot be referred to as an exception belonging to the new abstraction.

4.2. "Null" Exception Handlers

The implementation of any exception handler may be the `null` statement. The `null` statement is a statement that does nothing. A `null` handler masks its exception and performs no handling for it.

If `null` statements are to be used at all in exception handlers, they should indicate that an exception does not require handling. An example of a `null` handler is shown in Figure 4 for an operation that reads a file. The null handler causes an `End_of_file` exception to complete the operation, rather than allowing `End_of_file` to propagate to the invoker. In this case, it is entirely sensible for the handler to be `null`. It is only when `null` handlers are used for exceptions actually requiring handling that anomalies arise.

```
begin
   ...
   <read characters from a file>
   ...
exception
   when EndOfFile =>
      null;  -- end of file is expected
end;
```

Figure 4: Use of "Null" Handler

For example, the "`when others => null;`" construct has been seen repeatedly in one particular large program [3]. These statements are intended to construct "firewalls" in the program to prevent uncontrolled propagation of exceptions. However, no information is accumulated about the exceptions that are handled at those `null` handlers. It is unlikely that the reasons for their propagation will ever be found in the current implementation, because the `null` handlers prevent the exceptions from being noticed by the users of the program. If the firewalls had instead reported the exceptions that manage to propagate to that point, or at least reported that the "`when others`" handler had been invoked, the next revision of the software could attempt to fix the problems.

Definition 9: "Null" Handler Anomaly ::= If an exception handler *H* for exception *E* is `null`, then *E* is masked and not handled.

4.3. Ill-defined Non-specific Handlers

A `when others` exception handler is used to handle exceptions that are without another specific exception handler. The exceptions that will be handled by that handler are unknown—all that can be said is that they are none of the exceptions handled elsewhere in the frame. It is difficult to construct a handler on this basis, because there is no precise definition of the handler's applicability.

There is also no clear statement of what the `when others` handler should accomplish before completion. Since the exception being handled is not known ahead of time, the state of progress for the operation is also not known, and few assumptions can be made about the state of variables. Due to the uncertainty involved in the activation of the `when others` handler, it is used primarily to enforce simple properties, such as emplacing boundaries over which propagation cannot cross—firewalls.

Definition 10: Ill-defined Handlers Anomaly ::= Consider a frame *F* containing a `when others` exception handler. The maximal amount of information that can be brought to bear in the construction of that handler consists of 2 guarantees: the exception *E* that activates the handler was raised in frame *F* (perhaps through propagation), and *E* is not an exception listed in any other exception handler for that frame.

4.4. Mutual Exclusion within Exception Handlers

A data object may be shared by more than one active task in such a way that the object needs to be protected from simultaneous access by more than one task. The program may implement this protection by using *mutual exclusion* to limit the number of tasks updating the object at the same time. Any attempt to access the object without following the protocol used to enforce mutual exclusion is erroneous.

If an exception handler accesses the object without obtaining mutually exclusive access to it, then the handler is erroneous. If an erroneous exception handler accesses the object without mutual exclusion, then the object will most likely be left in a state that is inconsistent with its specification. If the assumption that exception handlers are executed less frequently than the "normal" sections of algorithms is true, then it may be the case that such an erroneous handler may be unintentionally included in an Ada program due to the difficulty of determining its presence during testing. This is not necessarily a difficult error to detect, because the program can be examined to ensure that all data objects needing mutual exclusion are accessed by following the appropriate protocol.

The more difficult question to answer about an Ada program's mutual exclusion is this: what state is the data object in when the exception handler is invoked? If the handler assumes that the object is locked when it is not, then the handler is erroneous. Similarly, the handler is erroneous if it assumes the object is unlocked. Unless careful program design has ensured that the exception handler for a frame will only be activated when the data object is in one state or the other, then the handler will probably be erroneous.

The difficulty in determining whether a given data object is locked or not lies in the fact that the answer to the question may change before the exception handler can act upon that answer. If the mutual exclusion mechanism reports that a data object is not locked currently, the handler cannot use this information effectively because another task may obtain access to the object before execution of the rest of the handler. Similarly, if the object is already locked, the handler can only wait until it becomes unlocked again—at which point it cannot assume the object is actually unlocked.

A example of a mutual exclusion problem concerns an *audit file* that is used to record the raising of exceptions. The audit file can be examined to determine the reasons for a program's completion. If only one audit file is kept for a program with multiple tasks, it is possible for the audit file to contain scrambled reports of exception raises if mutually exclusive access to the audit file is not maintained. This is because it is possible for more than one task to be handling an exception at the same time. If more than one handler writes to the file at the same time, the resulting report may be unintelligible.

Definition 11: Mutual Exclusion Anomaly ::= A data object *O* is specified as requiring a particular mutual exclusion protocol. The specification is broken if an exception handler *H* accesses data object *O* and does not obtain access appropriately.

4.5. Erroneous Dependence on Parameter Passing Mechanism

The Ada language reference manual [1] states in section 6.3 that an operation that depends upon a particular implementation for *in out* parameters is erroneous. The two standard methods for implementing *in out* parameters are passing the parameter by reference or passing it by copying. A parameter passed by reference is actually modified

during the course of the operation, while a copy of the parameter is modified in the other method. These two methods are effectively identical if no exception is raised. For a reference parameter, the end result is already stored in the original data object. For a copied parameter, the result is copied back into the original data object upon completion of the operation.

However, if an exception is raised, then the two methods may differ in their effect. It is up to the Ada compiler to enforce the copying out of a parameter passed by copying after an exception is raised. If that parameter is not copied, then the original value of the data object is unchanged, while a parameter passed by reference is partially modified (depending upon how far the operation had progressed before the exception was raised). This difference is the reason for stating that a program that depends on one method over the other is erroneous—if the compiler has implemented the parameter passing by the opposite method, the program will not operate correctly.

In addition, the program is not able to discern that an exception was raised in the operation if the exception is masked (i.e. it is not raised again). A masked exception raise causes normal completion of the operation, but the final value of the data object may still differ depending on the parameter passing method. To avoid a situation where the program assumes that the data object is still in a consistent state after the operation, the operation must either propagate the exception or must ensure that the object is returned to a consistent state. If the exception is propagated, the invoker can at least return the data structure to its original state; otherwise it has no knowledge that the object might be inconsistent.

Definition 12: Parameter Passing Anomaly ::= Consider a subprogram designated as frame *F*. If frame *F* is passed an array *A* as `In Out`, then *A* might be passed to frame *F* by reference or by copy. When a frame *G* invokes frame *F* by passing an actual parameter *B* as `In Out` formal parameter *A*, then frame *G* is non-erroneous if and only if its implementation yields the same result independent of the parameter passing method used for frame *F*.

4.6. Partial Exception Handling

A data object that is passed as a parameter to an operation may have assertions about its state before and after the execution of an operation. These assertions specify what the intended effect of the operation is on that parameter. If an exception handler does not satisfy an assertion about a data object, then it is erroneous. It is a *partial* exception handler, because it may perform some handling duties, but it has left the data object in a state inconsistent with its specification.

An example of a partial exception handler is one that does not set the value of an *out* parameter before completion of the operation [2]. *Out* parameters are intended to return information after an operation is completed; they have no value upon entry to the operation. If an *out* parameter is not set by the operation, then the value returned to the

invoker is indeterminate and definitely not the value intended.

Figure 5 shows an example of partial handling. There are two exceptions that can be raised by a division procedure, arithmetic_overflow and arithmetic_underflow. This procedure sets the result equal to the maximum representable real number if arithmetic overflow occurs—a reasonable response to the overflow exception. However, the procedure does not handle arithmetic underflow correctly. The result **z** is not set to a default underflow value of zero and the exception is masked. For arithmetic_underflow, the result of the division is not defined and the value of **z** is unknown upon frame completion.

Definition 13: Partial Handling Anomaly ::= If a frame *F* has a specification for a post-condition concerning a parameter *P*, and there exists a path in frame *F*'s exception handling control flow graph where *P* is not modified, then only the pre-condition of *P* is guaranteed after completion of *F* and the post-condition will probably be violated.

5. Coordination Related Anomalies

The anomalies in this section are the result of differing perspectives on how exception handling should occur. The differences make programs constructed using one pattern or model difficult to interface with programs constructed according to a different model. In some cases, the lack of coordination is induced by the insufficiencies of Ada's chosen exception handling model.

```
procedure division
    (x, y : in float; z : out float) is
begin
    z := (x / y);
exception
    when arithmetic_overflow =>
         z := MAXREAL;
    when arithmetic_underflow =>
         null;
end;
```

Figure 5: Partial Handling Example

5.1. Protocol Mismatches (e.g. Posix exceptions)

There are many validated compilers for Ada, and they are targeted towards a variety of operating systems. A discussion on the Internet involved the construction of a Posix interface for Ada. Posix is a standard for the UNIX operating system and details exactly what interfaces are to be provided to user programs. The debate concerned how exceptions are to be mapped to the Posix error codes.

For example, a read operation on a Posix file may be unable to continue reading because the end of the file has been encountered. This circumstance might be represented in Posix by returning an error code from the read operation instead of the number of bytes read. In keeping with the Ada convention, it is preferable to raise an exception such as Posix_end_of_file when this situation occurs. The debate centers around how the different Posix error codes are to be represented as exceptions.

The obvious choice from the Ada programmer's point of view is to make each operation declare appropriately named exceptions that are raised when the corresponding event occurs. A flaw in using named exceptions is that the Posix standard may change, rendering the programs based on those exception names invalid within the new standard. There may also be too many names if a unique name is assigned to each possible operating system error. Any individual Posix operation can have a number of error conditions associated with it; Ada operations based on the Posix operation might need a handler for each error condition, requiring a much enlarged set of exception handlers for that operation.

Another possibility that has been discussed is to have one exception that covers all Posix error conditions. This one exception would be raised if any of those errors occurs. The handler could then obtain the actual error code from a function before attempting to handle particular errors. This approach minimizes the size of handlers, because only the relevant errors require specialized handlers.

Definition 14: Protocol Mismatch Anomaly ::= Consider an object *A* that provides a service. An object *B* is a user of *A*'s service. Object *A* may have one method for exceptional conditions, while object *B* may have a different model. Integrating the two models is not necessarily straightforward or even possible.

5.2. Frame Resumption

Ada uses a *termination* model for exception handling—if an exception is raised in a frame, then either the exception is masked or propagated. In both cases, the frame is completed (or terminated). A different mechanism for exception handling is the resumption model—if an exception is raised in a frame, the frame may be resumed after the exception propagates to the invoker.

Although resumption can be simulated in the termination model by iteratively invoking the operation to be resumed until a successful completion occurs, resumption is not supported directly. This anomaly is somewhat different from the others in that no dangerous effects can immediately result from it. It is possible for an application to have a more complicated implementation due to the lack of resumption, and this can in turn result in a less stable program.

An example where a resumption model facilitates implementation is in a graphical menuing system. An initialization routine for the menu system is invoked once to create the appropriate objects needed. Then, to obtain menu selections interactively from a user, the routine `get_choices` is invoked.

The menus are hierarchical in that each top level choice may have several choices contained within it. The menu system manages the display of the appropriate menu and choices until the user selects a particular choice. At this point, an exception called `Choice_made` is raised and control returns to the invoker of the menu system. The program that is using the menu system can then process the user's selection.

After processing, the menu system can be resumed to continue managing user selections and displaying menus. This is where the resumption model facilitates the clarity of the implementation—a single statement similar to `resume menus` causes the menu manager to continue where it left off. Instead of 1) having to invoke the menu system by calling `get_choices` again, 2) paying the performance penalty of any initialization that `get_choices` performs, and 3) losing the state of the menu system, a single resumption could cause the menu system to begin processing again. The analogous implementation in the termination model needs to simulate resumption or construct the menuing system using a different approach. If some parts of the menuing system are pre-existing software components, then the second choice is not feasible.

Definition 15: Lack of Resumption Anomaly ::= Consider a frame *F* in which an exception *E* is raised. Frame *F* cannot be resumed by its invoker *I*, even if resumption would lead to a clearer or cleaner implementation.

5.3. Exceptions, Return Codes and Messages

At least three methods are used to indicate ''exceptional'' results from an operation: 1) actually raising an exception, 2) returning a value that is distinguishable from normal return values, or 3) passing a message describing the situation. Any or all of these approaches might be present in the same program; a value returned by one operation might cause its invoker to raise an exception, which might in turn cause another operation to send a message to a different region of the program. The ordering of return value/exception raise/message sending may not follow this sequence at all as a program can arbitrarily connect operations together that use any of the three methods.

The indiscriminate use of these three different mechanisms can cover the actual meaning of an exceptional condition with a shroud of supposedly useful activity. Since organization of the use of these methods is ad hoc, the programmer himself may soon forget the intended behavior.

Definition 16: Arbitrary Mapping Anomaly ::= Consider a case of three packages, *A*, *B* and *C*, where each package (respectively) uses an exception handling model for exceptional conditions, a return code model for exceptional conditions and a message passing model for exceptional conditions. Any combination of usage between these three packages involves a mapping of the condition handling model for the used package into the error handling model for the using package.

5.4. Interfaced Operations

Software that is written in languages other than Ada can be invoked from an Ada program through the pragma `Interface`. Ada operations are constrained to raise an exception when the specification of the operation is violated. Other languages do not necessarily have exception handling semantics. Operations in those languages must check constraints themselves, and handle violations of them appropriately. When an interfaced operation is called from Ada, no exception can be propagated back to the Ada program, even if the exceptional condition is detected by the operation.

Figure 6 depicts an example of the interfacing problem. The interfaced operation is a function written in the C programming language. The C function does not maintain the constraints on the variable passed to it. When the Ada program attempts to use the variable, it is a larger number than its declaration allows. There is no default method for passing information about variable constraints to the interfaced code. Even if a mechanism for passing this information existed, the C function would probably be a general library function that was not designed to make use of the information.

Definition 17: Interfacing Anomaly ::=

If a routine *R* implemented in a language other than Ada is invoked from an Ada program, then *R* cannot raise exceptions across the interface. If a constrained Ada variable *V* is passed to routine *R* through the interface, *R* has no knowledge of constraints on *V*. If *R* violates constraints on *V*, the Ada program cannot be alerted by the raising of an exception.

6. Coupling Related Anomalies

The anomalies in this section occur due to the reliance of one part of a program upon another part of the program. Some dependencies are unavoidable in the construction of a large program. Coupling anomalies arise when an expected service can

```
with text_io; use text_io;

procedure main is
  type day is (mon, tue, wed, thu, fri, sat, sun);

function tomorrow(d : day) return day;  -- tomorrow just increments argument.
pragma interface (C, tomorrow);  -- in reality, tomorrow would be a useful C function.

  today : day;

  package day_io is new enumeration_io (day);
  use day_io;

begin
  put ("Today is sunday.");
  new_line;
  put ("Tomorrow is ");
  put (tomorrow(today)); -- c routine goes past sunday.  no exception is raised.
  new_line;
end main;
```

Figure 6: A "C" Function used in Ada

no longer be provided or when it is provided incorrectly.

6.1. Assumption Coupling Issues

A common practice in programming is to isolate an algorithm that is used frequently and place it in its own operation. Usually the programmer finds that he has used the algorithm in multiple places in slightly different forms. After creating this new functional abstraction, the program will usually be shorter and easier to understand than before.

The new operation that has been created has pre-conditions that must hold before it can be invoked, and it has post-conditions that it ensures before completion. The post-conditions are, in a sense, the effect of the operation, and are the reason it is invoked. However, a particular exception raise may cause the operation to fail to ensure one of the post-conditions before it is completed; this has already been discussed under the guise of partial handling. The post-conditions of that operation might be intended to partially guarantee the pre-conditions of a subsequent operation, in which case the subsequent operation's pre-conditions are violated.

Assumption coupling is one way that partial handling can effect the rest of the program. The actual problem exists in the exception handler for an invoked operation, but the effects of that problem are felt after the invoked operation has completed.

Definition 18: Assumption Coupling Anomaly ::= Consider the case of two frames *A* and *B*, where *A* is invoked near the start of *B*. The pre-conditions for frame *B* may include an assertion *X* that is one of the post-conditions of frame *A*, since *B* invokes *A*. Frame *B* is thus coupled to frame *A* by its need for the services of *A*. If assertion *X* is not guaranteed by frame *A* due to exceptional control flow, then the failed post-condition *X* becomes a failed pre-condition for frame *B*. The effect of an erroneous implementation of *A* has propagated to the implementation of *B*.

6.2. Task Coupling Issues

Tasks are said to be coupled if the correct performance of one task depends on the correct performance of another task. If many tasks in an application program are coupled, then it is possible for a ''House of Cards'' failure. This calls to mind the precariously perched tower of playing cards—the situation is similar for the tasks. If one frequently used task completes due to an exception, then the tasks communicating with it are unable to obtain its services. These tasks may in turn complete due to the TASKING_ERROR that is raised when they attempt to rendezvous. The extreme case is that several tasks complete due to the initial task's completion and the system is brought to a standstill while these tasks are restarted, or worse, system failure could occur.

Definition 19: Task Coupling Anomaly ::= If a task T_1 requires the services provided by task T_2, T_1 is said to be coupled to T_2. If T_2 completes unexpectedly, TASKING_ERROR is raised in T_1 at the point of rendezvous with T_2. If this raised exception causes completion of task T_1, then all tasks coupled to T_1 will encounter a TASKING_ERROR exception when they attempt to rendezvous with T_1.

References

1.: A. N. S. I. Inc., Reference Manual for the Ada Programming Language, February 17, 1983.

2.: C. Howell and D. Mularz, Exception Handling in Large Ada Systems, *Washington Area Ada Symposium*, 1991.

3.: C. A. Koeritz, A Non-Disclosure Agreement was Signed in Reference to this Information.