SPECIFYING PROGRAM SEMANTICS PRECISELY AND HIERARCHICALLY: CONSEQUENCES AND THE DEVELOPMENT OF OZ

C. A. Koeritz
J.C. Knight

Computer Science Report No. TR-93-12
January 26, 1993

Specifying Program Semantics Precisely and Hierarchically: Consequences and the Development of OZ[†]

C. A. Koeritz
J. C. Knight

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

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

Abstract

The proper handling of exceptional conditions is important in computing systems that are intended to be dependable. The Ada language is used for programming such systems and provides *exception* semantics as a means for handling exceptional conditions, but the mere semantics do not guarantee any properties of dependability. Numerous unusual situations (called anomalies here) can exist in the exception handling portions of an Ada program that cause the program to diverge from its specification, often in a completely unacceptable manner.

To increase dependability in existent Ada programs, anomalies must be located and removed from them. When implementing new Ada programs, anomalies should be precluded from the system by applying exception handling design principles during program construction. Initially it was believed that these two goals (detection and prevention of anomalies) were sufficient to substantially increase dependability in Ada programs, but now it is felt that the anomalies of Ada arise due to basic problems with the nature of exception handling itself. It is asserted that exception handling is a fundamentally flawed paradigm and that it is not sufficient for the needs of dependable computation.

This paper urges a revision of the notions upon which exception handling is based. A new and fundamentally different view of the problem domain currently addressed by exception handling techniques is presented. A model based on the *consequence* is discussed as an implementable and useable method for addressing the needs of dependable computation. A consequence's intent and causes are more precisely specified than for exceptions, and the consequence is a more general and flexible mechanism than the exception.

The consequence model forms the basis for the *OZ* specification notation. This notation can be added to Ada programs to guarantee properties of their behavior, even if certain classes of anomalies are present in the implementation. Augmenting an Ada program with accurate OZ specifications also embeds important information in the program's specification that can be used later for verification. Using OZ, certain properties of dependability can be guaranteed a priori; exception handling only begins to afford similar assurance after a difficult and potentially impossible verification of the ad hoc programming employed to handle exceptions.

© and last revised January 21, 1993

Acknowledgements

The assistance of the MITRE Corporation is gratefully acknowledged. In particular, Chuck Howell, Diane Mularz and Gary Bundy helped to define the various problems in Ada and offered numerous helpful suggestions. This work sponsored in part by the MITRE Corporation and in part by the Virginia CIT under grant number INF-92-002.

Table of Contents

1. Introduction 1

2. Previous Work 3

3. Ada Exception Handling Semantics 6

4. Anomalies 8

5. Axioms 10

    1. Taxonomy 10

    2. Static Analysis 11

    3. Guidelines 11

6. Dual Graph Model for Ada 12

7. The Fundamental Nature of Exception Handling 13

    1. Operations and their Consequences 14

    2. Hierarchical Program Design 18

    3. Verification 20

8. OZ: A Specification Notation 22

    1. OZ Description 22

    2. Toolset 25

9. OZ Examples 27

    1. Anonymous Exceptions 27

    2. Uncontrolled Propagation 27

    3. Tasking Deadlock 27

    4. Using OZ to Enhance Dependability in Existing Ada Programs 28

    5. An Example OZ Specification & the Resulting Ada Program 31

10. Conclusion 38

References 39

Bibliography 41

Figures

Figure 1: Propagation Delay Causing Deadlock 9

Figure 2: The Essence of an Operation 15

Figure 3: Stack Operations in terms of Consequences 17

Figure 4: An Example Operation 23

Figure 5: An Example Zone 24

Figure 6: Hierarchical Ada Program 29

Figure 7: Read_brightness Operation 33

Figure 8: Ada & OZ Toaster Specification 33

Figure 9: Ada & OZ Toaster Implementation 34

Figure 10: Full Ada Toaster Implementation 34

Figure 11: Safety Check Specification in Ada & OZ 35

Figure 12: Safety Check Implementation in Ada & OZ 36

Figure 13: Full Ada Implementation of Safety Check 37

SECTION 1

Introduction

Insuperable problems have been encountered in the Ada language's exception handling semantics. The problems are collectively called *anomalies*, because their presence can cause a program to exhibit unexpected and unusual behavior when an exception is raised. If anomalies exist in an application, then some desired properties of dependability cannot be guaranteed. In an effort to increase the dependability of programs developed in Ada, it is important to uncover the anomalies, discover their causes and develop solutions for them.

To lay the foundation for understanding the problem, the previous work in exception handling is reviewed in Section One. Section Two reviews the semantics for exception handling in the Ada language to offer some terminological background for the work. To highlight the need for a solution, Section Three describes several of the anomalies of Ada exception handling and their possible effects on Ada programs. Section Four outlines the approach to detecting and eliminating anomalies from Ada programs— by employing a set of axioms that describe the anomalies in terms of Ada's exception handling semantics, precise causes can be ascribed to each anomaly and it is possible to reason from the causes to a solution. Section Five describes a bipartite graph model that represents control flow in an executing Ada program when performing exception handling.

To fully understand the nature of exception handling, it is necessary to ascertain the programming requirements addressed by exception handling. Several questions help guide the search for these requirements:

- What exactly is an exception?
- Why are exceptions a concern at all?
- What are the underlying programming needs that exception handling attempts to address?
- What are the limits of exception handling?

Although answers have been proffered in the past, these are usually within the context of a specific language or are at best constrained within existing notions of exception handling. A goal of this research is to discover the ''atom'' of exception handling—to uncover the essential characteristics of a flexible exception handling mechanism that assists in the verification of programs using it.

Exception handling is used as a major design technique in many programs, and semantics for exception handling are provided in several programming languages. This in itself indicates that exception handling fulfills some need in the software, whether an imagined or real one. It is asserted here that the need for exception handling is indeed real, although the relative merits of existing exception handling models are debatable.

Further, it is asserted that although there is a need for exception handling, there are greater needs than current exception handling models can fulfill, among them the need for verifiability, dependability, and allowance for hierarchical design.

While the set of anomalies discovered in Ada are an instance of insufficiencies in one exception handling model, it is asserted that there is a need for a model that can be used for the purposes of conventional exception handling, but which does not suffer from what are now perceived as fundamental problems with the nature of the exception handling paradigm itself. The *consequence* model is proposed to supplant existing exception handling models. The consequence model can be used to implement any of the previously proposed exception handling models—including the termination [1], resumption [2] and replacement models [3] if desired. In addition, a failing of many exception models is an inability to handle exceptions within expressions adequately—consequences integrate expressions seamlessly with statements to grant more flexibility than other models. It is asserted that consequences capture the concerns of exception handling precisely and completely.

A specification notation called *OZ* is based on operations as they are defined in the consequence model. OZ is offered as a means of specifying robust and verifiable program behavior. Many of the Ada anomalies are prohibited from appearing in a program specified with OZ as a matter of design. This assurance provides opportunities for verifying Ada programs that have been augmented with consequence specifications.

Section Six discusses the consequence model in terms of the fundamental nature of exception handling. In Section Seven, the OZ specification notation is described as a method by which the consequences of operations may be specified and verified. A plan of work is outlined in Section Eight to show how the goals of developing OZ and using it within Ada programs will be accomplished.

SECTION 2

Previous Work

Goodenough ordered the previously cluttered field of exception handling by outlining exactly what exceptional services are often desired and then detailing semantics that provide them [2]. He defines an exception as a condition that must be brought to the attention of an invoker. His model is *resumption based*—an operation may signal an exception but then be resumed again afterwards. He specified three cases for resumption:

- always resuming,
- never resuming, and
- sometime resuming, dependent upon the invoker's situation.

Goodenough's model for placement of handlers allows a handler to be attached to any program statement that can raise an exception, including expressions. This is general, but suffers from the problem of recursive inclusion of handlers within handlers ad infinitum; if an exception handler can raise an exception, then a handler may need to be attached to it to guarantee that the operation meets its specification. If the handler's handler can raise an exception, then it in turn needs a handler. Each new handler deepens the level of nesting within the operation. Not only are programs constructed in this way not clear, but it is also difficult to guarantee that the specification is met by the rat's nest of handlers.

Cristian simplified the implementation and analysis problems by limiting his model to a *termination* semantics [4, 5]. In the termination model, an operation that raises an exception is completed by that raise and cannot be resumed. Cristian defines an exception as an operation's response when its specification cannot be met, either due to a violation of the operation's pre-conditions or an inability to satisfy its postconditions. His definition for a program specification includes information regarding its exceptional specification as well.

Cristian uses backwards and forwards predicate transformations of the specification along with his precise definitions of operation semantics to prove certain properties of programs, such as robustness. His approach may not be very easy to implement however, because these predicate manipulations require a great deal of domain specific knowledge on the part of the verifier, who is in this case a person.

Yemini proposes a *replacement* semantics for exception handling that can simulate both the termination and the resumption model [3]. Her approach involves replacing the result of an operation that raises an exception with the handler for that operation, in effect replacing the operation's result with its handler's result. This approach amounts to calling a function when an exception is raised and either 1) using the function's result in place of the exception raise itself, or 2) using the function's result in place of the operation's result. The first case emulates resumption semantics and the second termination semantics.

The replacement model does not address the infinite recursion of handlers problem, however; the handlers replacing the raised exception may raise exceptions also. Also, it is not clear exactly how much program size will increase due to the addition of special purpose functions for performing the handling; since replacement is effectively a function call, each handler function must be implemented and included in the program. Finally, it is not clear whether the replacement model is significantly more powerful than a language that implements exception handling only through function calls.

Clu provides exception handling semantics that implement a one-level termination model [6]. This model enforces declaration of all exceptions that an operation can raise, and does not allow an invoker to re-raise an exception unless it has been declared. This model avoids some problems with propagation, but allows others; instead of rigidly enforcing a rule that all possible exceptions are handled, Clu allows operations to fail if they do not handle an exception. The failure exception is thus an implicit exception of all operations. This concept is useful, but can lead to a situation where an unhandled failure completes the Clu program.

Luckham and Polak detail a method for verifying Ada programs that enforces:

(1) explicit *declaration* of all exceptions that can be raised by an operation [7].

(2) attaching an *assertion* to every exception handler detailing the pre-conditions for its handler activation.

If the declaration is correct, then the set of all exceptions that might be raised by an operation can be constructed. The declaration part of the specification cannot be constructed in a straightforward manner, however. In Ada, for example, there are five pre-defined exceptions that can be raised by numerous primitive Ada operations (e.g., addition), and it may be quite tedious to decide whether an operation can raise one or more of these.

If handler assertions are specified accurately, then the pre-conditions for activating a particular handler are known. This is not straightforward either, as these assertions are selected by the programmer and no process for constructing them is outlined. Indeed, no definition is provided for an Ada program specification either, so it is unclear what goals the handlers must achieve. Given accurate declarations and assertions, and assuming there is a program specification, Luckham and Polak provide a proof technique to guarantee that the Ada program meets its specification.

Black was one of the first to take issue with the very existence and use of the exception [8]. He asserts that the exception is as powerful and dangerous as the goto statement. While the language semantics in most languages structure exception handling more than they structure gotos, Black was the first to recognize that there are problems with many aspects of exception handling methods.

He makes the point that the exception is not a well-defined concept. This is a flawed argument, because the exception has been defined well by several authors (including Cristian), and the exception serves a necessary role in program construction. Without it, there is no language defined mechanism for signalling that an operation's preconditions have been violated, nor for signalling that the postconditions cannot be met. In languages without exception handling, the result returned by an operation must be checked to ensure that the operation succeeds.

Black does provide a substitute mechanism for the exception. Operations may return *oneof* a number of different results, some of which essentially denote that an exception has been detected. He claims that all programs can be written just as elegantly without exceptions by using this mechanism. The first part of the claim is guaranteed by Turing equivalence; any program with exceptions can be rewritten without exceptions. The second part of the claim, elegance or ease of use, is not justified, because Black provides very little beyond current exception handling techniques to manage his *oneof* results. In the past, operation results have served to signal that exceptional situations have occurred, but there is no guarantee that the programmer will check for the exceptional result. At least exceptions provide a way to enforce that the program either handles the exception or is halted.

Cui and Gannon claim that by associating exception handlers with data types, program structure can be made clearer and programs can be made smaller [9]. Their semantics add parameterized exception handlers to the definition of Ada data types by including pre-processor directives for declaring, raising and handling exceptions associated with the type. Their approach seems sound, but it is unclear how a data type's exceptions are handled other than through the pre-processor directives; they do not detail the interaction involved if a situation arises where it is desired to handle a data type's exception using Ada's own exception handling semantics. Also, since their approach concerns only ''implementation insufficiencies'' as described by Black [8], it is not clear how generally applicable data oriented exception handling is.

Some of the papers in the field of exception handling attempt to verify properties of exception handling programs by using precise semantic definitions and a program specification. In every case, however, the verifications are only attempted on a subset of an existing language, or are attempted on a small set of pseudo-language semantics. None of these papers addresses the problem of verifying the properties of large programs for the full semantics of an actual language, such as Ada. This is due in part to the complexity of Ada and the complexity of arbitrarily structured programs. For verification to have a real possibility of success, then one or both of these complexities must be reduced.

SECTION 3

Ada Exception Handling Semantics

Although the Ada Language Reference Manual [1] is fairly precise, its form and content do not make its implications apparent. Many notes about exception handling are strewn throughout the LRM, rather than being contained in the exception handling section. By grouping all language defined information regarding exception handling in one readable document [10], numerous rereadings of the LRM can be avoided. For clarity, a summary of the Ada exception handling semantics is presented here.

The Ada semantics define that an exception must be declared within a *scope*, much as variables are. The scope of an exception may be:

- a *procedure* or *function*, both of which are called *sub-programs* in Ada,
- a *block*, the smallest 'part' out of which sub-programs are constructed,
- a *task*, the construct for concurrency in Ada, or
- a *package*, an object-based mechanism for implementing abstractions composed of tasks and sub-programs.

Exceptions are *raised* to signal that an exceptional condition has been detected. Exceptions are raised in a *frame*, which is the representation during execution of one of the four scopes described above. When a frame is invoked, it either completes normally or it raises an exception.

A group of exception *handlers* may be attached to the bottom of any frame. A handler is activated when its exception is raised in the frame. If the handler itself does not raise any exception, then its own exception is said to be *masked*. A masked exception raise is treated by the invoker exactly as if the invoked frame completed normally. A handler for the special name `others` handles any exceptions that have no other exception handlers in the frame.

An exception may *propagate* to the invoker of a frame. Propagation means that the exception is raised again <u>inside</u> the invoker (at the point where it invokes the frame propagating the exception). Propagation occurs if one of the following is true for the frame in which an exception is raised:

- the exception has no handler in the frame,
- the handler raises the exception again, or
- the handler raises another exception.

Note that in the third case, the propagated exception is different from the one raised initially.

These semantics are not extremely complex, although certain of their interactions with other aspects of the language can be complex or counter-intuitive. It is interesting that such a small set of semantics, along with a few incongruous special cases, is capable of generating the wide variety of problems that have been encountered.

SECTION 4

Anomalies

A catalog of the known Ada exception handling anomalies is compiled in [11]. The cataloging effort provided a great deal of insight into why programs might be built incorrectly when exception handling is used as a major design policy. To date, twenty anomalies have been discovered.

Some of the anomalies are the result of peculiar nuances of the Ada semantics. In Section 11.4.1.9 of the Language Reference Manual, exception propagation is specified as being delayed when dependent tasks have not completed [1]. This rule causes a program deadlock when:

(1) a task is instantiated in a frame,
(2) an exception raise circumvents rendezvous with the task, and
(3) propagation is delayed awaiting the task's completion.

The resulting anomaly is called *tasking deadlock due to delay of propagation.* The example in Figure 1 shows a program where a tasking deadlock is present.

Another anomaly resulting directly from the Ada semantics is the *anonymous exception.* Exceptions are visible within the scope of their declaration, and obversely there are scopes within which an exception is not visible. If an exception is propagated to a scope where its declaration is not visible, then that exception becomes anonymous. An anonymous exception stays anonymous even if it is propagated back into a scope where the original name is visible. Anonymous exceptions plague Ada designers, because an anonymous exception cannot be handled using its own name.

Other anomalies result from the nature of a language's chosen exception handling model. Ada's model incorporates exception propagation as the method for signalling the invoker of an operation that an exception has been raised. This design choice means that an invoked operation may raise an exception in the invoker to signal an exceptional exit of the operation. A number of anomalies result from the Ada interpretation of exception propagation. These anomalies subsist primarily on this fact: although exceptions are stated as *existing* when an Ada operation defines them, they are not declared as *propagatable* in the operation's interface. In other words, an operation's exception handling behavior is not fully specified in Ada, because some exceptions can be propagated to the invoker without warning.

This is a serious deficiency if Ada is to be used for applications requiring dependability. It is not easy to determine the set of exceptions that can actually be propagated. If an operation's invoker is to handle the exceptions that it can propagate, then the implementation of the operation *and all operations that it invokes* must be examined. This situation is unbearable for the programmer and does not assist in the

```
with Text_Io; use Text_Io;

procedure Deadlocking_procedure is
  -- This exception's raise will cause the deadlock.
  exception_whose_propagation_is_delayed : exception;

  -- Sleeper is a dependent task of the Deadlocking_procedure.
  task Sleeper is
    entry Go_Ahead;
  end;

  task body Sleeper is
  begin
    accept Go_Ahead;
    delay 10.0;
    put_line("After delay in Sleeper.");
  end Sleeper;

begin
  -- initial computations performed here...
  -- eventually, this exception is raised:
  raise exception_whose_propagation_is_delayed;

  -- the exception's raise causes this task entry not to be reached.
  Sleeper.Go_Ahead;

exception
  when exception_whose_propagation_is_delayed =>
    put_line("The exception is handled by Deadlocking_procedure.");
    -- The next raise would propagate the exception, but since Sleeper is not
    -- complete, the propagation is delayed.  Sleeper won't complete, since it
    -- is awaiting a task entry.  Thus, deadlock results.
    raise;
end Deadlocking_procedure;
```

Figure 1: Propagation Delay Causing Deadlock

program design process.

A related anomaly in Ada is the *uncontrolled propagation* of an exception. Since Ada does not enforce declaration of the propagatable exceptions, it is possible that an exception can propagate beyond the point where it provides meaningful information to the program. An uncontrolled propagation can actually complete the program by propagating the exception back to the root control program. An exception handler for `others` can keep the exception from propagating beyond a certain point, but it can only perform the most general handling of the exception.

Ada's exception handling semantics allow the inclusion of anomalies in programs. These anomalies are not only hard to locate and eliminate from the program, they are also dangerous because generally they break the program's specification. The difficulty encountered in using the current Ada semantics appropriately indicates the need for a different exception handling solution than Ada provides.

SECTION 5

Axioms

A set of axioms expressing the exception handling semantics of Ada can be used to state the precise reason for each anomaly's existence. The axioms also lead directly to the three important research offshoots of: 1) an exception handling taxonomy, 2) algorithms for anomaly detection through static analysis of Ada programs and 3) guidelines for creating anomaly-free Ada programs. These are introduced below.

1. Taxonomy

There are several underlying causes for the anomalies. The most general of these causes are:

- Exception handling is not easy.

  Programmers often use ad hoc methods to handle exceptions, rather than thoroughly considering how a raised exception may affect the requirements of the program specification.

- Exception handling is too powerful.

  Although normal program execution follows a well-defined ordering of events, the raise of an exception can perturb this order. An arbitrary number of operations may be completed (aborted prematurely, in this case) before a handler for the exception is located.

- Exception handling is flawed.

  Numerous unquestioned assumptions are often encoded directly into language semantics for exceptions. A few of the more pandemic assumptions are:

  (1) a raised exception represents an emergency situation,
  (2) exception raises are rare,
  (3) exceptions are always treated exceptionally, and
  (4) sequential control flow is the general rule when not handling exceptions.

  These assumptions are justified for many cases, but not for all. In situations where they are not justified, the effects on the program may range from extensive extra programming to get a desired effect all the way to programs breaking their specifications due to a lack of knowledge about, or an incorrect interpretation of, an assumption.

2. Static Analysis

The first hypotheses regarding the Ada anomalies suggested that static analysis might be able to find anomalies that were present in program implementations and report them with "red-flags". MITRE has constructed an analyzer capable of detecting several of the anomalies, such as anonymous exceptions, uncontrolled propagation, and ill-considered use of `others` handlers. The approach taken by MITRE uses the Diana intermediate representation for Ada [12] to provide access to a disambiguated, compiled form of Ada programs. Diana allows an abstract syntax tree for the entire Ada program to be traversed and examined. By searching the tree of importations making up the whole program, anomalies can be located and reported.

Many problems can be located this way, but others might escape notice. Since the association of handlers to exceptions is dynamic, some programs may require a full Ada simulator in order to ensure that all combinations of exceptions with handlers are examined. It is preferable to devise a technique that is not only capable of finding problems in programs, but is also capable of guaranteeing certain program properties. Instead of a dual compilation and anomaly inspection being necessary for every change in the program implementation, a single compilation phase is advocated—during this phase, necessary exception handling structures should be added to the program to guarantee certain aspects of the behavior during execution. A methodology for this alternate approach is suggested in Section Eight in the sequel.

3. Guidelines

A set of guidelines for Ada program construction can preclude certain anomalies from the program implementation. If the existence of an anomaly A depends on interactions between three features F_1, F_2 and F_3, then it is fairly clear that the anomaly can be prevented by disallowing either F_1, F_2 or F_3. The guideline for anomaly A is in a form that explicitly describes how to keep these three conditions from being present at the same time in a program. Each guideline follows one of two forms: either combinations of features are warned against or an anomaly-free pattern for combining certain features is provided.

SECTION 6

Dual Graph Model for Ada

A graph model for control flow during Ada exception handling can assist in program analysis [13]. The model is based on the idea that there are dual control flow graphs in an Ada program: one graph for ''standard'' control flow and the other for ''exceptional'' control flow. While transitions from the standard to the exceptional graph are fairly comprehensible, some of the possible transitions from the exceptional control flow graph back to the standard graph are not intuitive or obvious.

The graph model can be used to determine certain program properties on sight, because some anomalies have a visible ''signature'' that can be observed in the graph. For example, the uncontrolled propagation anomaly arises when an exception is not handled appropriately within a vicinity where its meaning is known. The extreme example of uncontrolled propagation is when a program completes because no handler exists for that exception in the entire chain of operation invocations. In the dual graph model, the presence of an uncontrolled propagation becomes obvious; there is a long sequence of links between the exception handlers in the exceptional control flow graph.

The graph model makes an important characteristic of Ada immediately apparent; programs are schizophrenically split into normal parts and exceptional parts, rather than being constructed in a unified manner. The fact that there are two separate and distinct control flow graphs means that a programmer must actually create <u>two programs</u> that interact by switching back and forth between the two graphs. This is not the most obvious realization one can have about Ada or its exception handling semantics, although the dual graphs are built directly into the language as <u>the</u> major control structure.

SECTION 7

The Fundamental Nature of Exception Handling

The *exceptional domain* for an operation is the space of operands for which the operation cannot yield a ''normal'' result, whereas the *standard domain* is the operand space for which the operation does yield a normal result [4]. The most basic cause for the existence of an exceptional domain is that an algorithm is usually only a partial function over its operand space; limiting operands to the algorithm's standard domain ensures that the algorithm produces a normal result, but when the algorithm is invoked in its exceptional domain, the result is not defined.

Exceptions cover the ''holes'' of the exceptional domain. Rather than producing incorrect results without warning, or enforcing a fail-stop policy that aborts the entire program, the operation can raise an exception. Exception handling is thus an attempt to return a program to the ''normal'' space of operands and results.

While this seems to be a fairly well-accepted definition of exceptions and exception handling, the exception handling models that are created with this definition in mind are insufficient for the purpose of creating large programs with useful properties such as dependability or robustness. Models developed using this definition are insufficient for three reasons:

(1) Exception handling models always suffer from the the assumptions noted earlier in the exception handling taxonomy discussion. These are built into the implementation of exception handling in existing languages in an attempt to limit the complexity of the exception handling semantics without hindering effective programming. The assumptions built into a particular language must be understood before programming in that language, or there is a high likelihood that the programs created will perform differently than expected.

(2) Anomalies can exist in current program implementations. Programs containing anomalies cannot be proven dependable, simply because they may exhibit behavior that breaks their specification. The detection and elimination of anomalies are not simple activities, and they cannot be completed for non-trivial Ada programs unless certain restrictions to the semantics are enforced and additional semantics are added.

(3) The exception model is fundamentally flawed and imbalanced as will be detailed below.

To substantiate the statement that exception handling is fundamentally flawed, it is necessary to consider the limitations encountered when using a model based on exceptions. First of all, the exception model contains the idea that an operation provides a standard service and that there are some ''abnormal'' services that are represented by exceptions. But the attribution of results as standard or exceptional is *not intrinsic to the operation*; these words denote viewpoints rather than facts. From an objective point of view, all that one can say is that an operation can be invoked on some operands and may have several outcomes. The labelling of these outcomes as standard or not is irrelevant to their existence or their usage. Normal is in the eye of the beholder and should not be built into a language.

Second, the exception model assumes that there are only two basic categories: normal and exceptional. This is a simplifying assumption, in many cases, because the operation is invoked to achieve a particular purpose, and failing that, raises an exception. But the exceptions that are raised are not necessarily of the same character; they are not necessarily each as "exceptional" as the other. For example, there are no language defined methods for treating an end-of-file exception differently than an out-of-memory exception in Ada, although they denote very different situations. In the case of Ada, both exceptions terminate the frame in which they are raised. Instead of the language providing semantics to describe clearly what each exception means, programmers must resort to a "bag of tricks" approach to handle them differently.

The third fundamental problem with exception handling is that when languages attempt to emphasize generality and expressive power in the exception semantics, it is difficult to prove properties for a program that uses exception handling as a design methodology. This is relevant to the Ada anomalies, because Ada is a complex language and most aspects of it are somehow integrated with exception handling. Unless all possible control flows in an Ada program are analyzed, there is no guarantee that the program is free of anomalies. There is no full specification of what exceptions can be raised, nor is there one for why they are raised; this effectively eliminates the possibility of program verification.

These three fundamental flaws cannot be expunged from exception handling because they are a part of its basic nature in every existing model. It is submitted that a model should not be changed when it has fundamental flaws—it should be abandoned.

1. Operations and their Consequences

Exception handling is an aged paradigm that seems to create more difficulties in the design and production of dependable systems than it alleviates. A new definition is offered here for a general program construct called the *operation* that is appropriate for performing the same tasks as current exception handling models. However, this definition does not incorporate the assumptions about exceptions that are perceived as the major source of exception handling anomalies. Additionally, this definition is suitable and useful in the context of program verification—programs based on operations are clear about the expected domain of applicability and specify precisely the possible consequences of the operation.

The notion of an operation is recursive—an operation can represent primitive computations provided by a particular programming language or computer, or it can be used to represent complex computations that are composed from multiple lower-level operations. Operations have both a specification and an implementation, although clearly the implementation of primitive operations may not be available for inspection.

A set of definitions will now be presented to clarify the meaning of an operation in the consequence model.

Definition 1: Element ::= A member of a data type.

Definition 2: Operand ::= An element passed as a parameter.

Definition 3: Operation ::= a computation that is applied to some number of operands. It concludes with one and only one of the *consequences* listed in its specification.

The *consequence* is the semantic construct that replaces the exception. The consequence details exactly what the operation accomplished upon its conclusion. Operations are connected through their consequences—a consequence of one operation may pass its result as an operand to a subsequent operation. This connection will only become active if the first operation concludes by that particular consequence.

The basis of the operation is a data-flow model—operations only activate when all of their operands become available. Some operations, most notably numeric, have a precedence associated with the operands specifying the order in which they are gathered. In the absence of explicit control flow to compute operands, the precedence is used to order the computations of operands unambiguously.

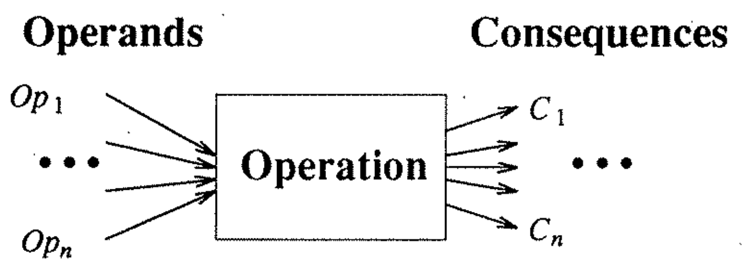

Figure 2: The Essence of an Operation

The flowing data is not only composed of discrete types, but also consists of information about changes to the state of operands if an operation modifies them. Such changes in state must be explicitly mentioned in the specification for the operation.

The choice of a data-flow model as the basis for operations in the consequence model does not limit applicability because control-flow can be simulated if necessary. A data token called the *control point* can be added as an operand for sequencing operations that have no obvious data-flow connections. The control-point ensures that operations are applied in a particular order. Since control-flow oriented languages like Ada, Pascal, C, and so on are unable to eliminate the data-flow that occurs in expressions (indicating that data-flow must be addressed in control-flow models also), and since the data-flow model can simulate a control-flow model, this presentation relies on a data-flow model.

Operands have already been discussed as the data flowing into an operation. The consequence embodies any data that flows out of the operation after it is applied to its operands and has completed. An operation's consequences are defined as follows:

Definition 4: Result ::= information that may flow away from the operation. Results are specified by assertions that describe:

(1) changes in the state of the operands after an operation concludes,
(2) an element produced by the operation, or
(3) both state changes and a produced element.

Definition 5: Outcome ::= a path for data flow out of the operation. The outcome distinguishes the nature of the result in terms of the possible operands to the operation.

Definition 6: Consequence ::= the 2-tuple consisting of the result computed by the operation and the outcome on which the result travels.

Different outcomes arise because an operation's domain, the operand space, is partitioned into equivalence classes. The equivalence classes characterize the combinations of elements that must be treated differently from other combinations due to the nature of the operation's implementation, or due to peculiarities in the definition of the element types. Each outcome represents the possibility that an algorithm may have to perform differently for elements in these different classes. For example, the stack data structure has three equivalence classes: stacks are empty, non-empty, or full. While a stack push operation can add an element to an empty stack and a non-full stack, it cannot add to a full stack. Likewise, a stack pop operation cannot remove an element from an empty stack, but may from the other two kinds.

The three common operations on stacks are depicted in Figure 3. The result specifications are based on an array-like notation for a stack. The '+' operation adds a new element to the stack, the '-' operation removes an element from the stack and the brackets access an element on the stack at a particular position (usually the top). Size is the current number of elements on the stack.

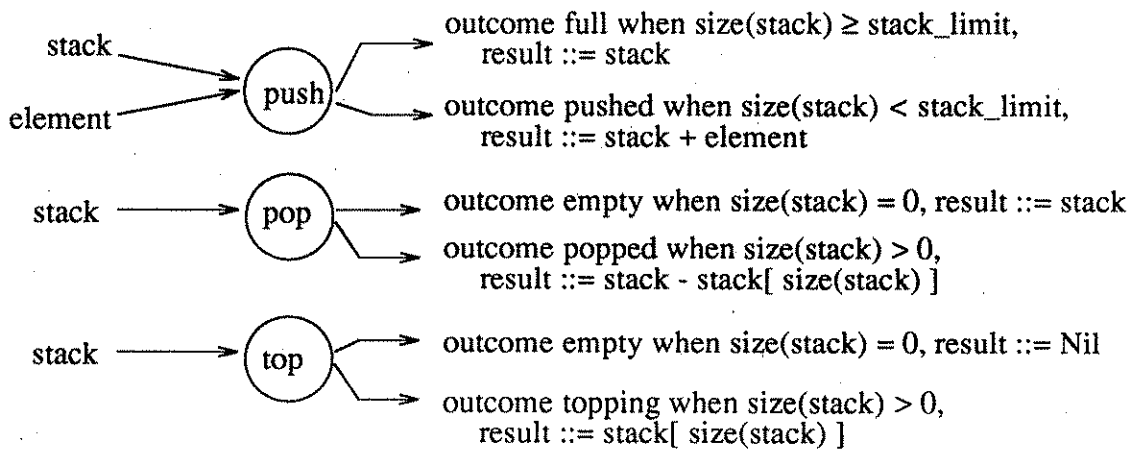
stack
element
push
outcome full when size(stack) ≥ stack_limit, result ::= stack
outcome pushed when size(stack) < stack_limit, result ::= stack + element
stack
pop
outcome empty when size(stack) = 0, result ::= stack
outcome popped when size(stack) > 0, result ::= stack - stack[size(stack)]
stack
top
outcome empty when size(stack) = 0, result ::= Nil
outcome topping when size(stack) > 0, result ::= stack[size(stack)]

Figure 3: Stack Operations in terms of Consequences

The fact that the operand space is partitioned ensures that operations are total over their operands, rather than being partial functions, because each partition is represented by one outcome for the operation. This is the essence of a robust operation—the operation has a response defined for every combination of operands [4]. The outcome is defined in terms of the equivalence class that it represents, because all of the members of an operand class cause the operation to pass a result down that outcome's path.

The consequence model posits a balanced view, because there are no adjectives like ''normal'' prejudicing the operation's definition; an outcome describes only the equivalence class of the operands without enforcing any interpretation of the result passed along that outcome. The only implication that can be arrived at about an outcome is that if the operands fall into that particular equivalence class, then the result will travel along that outcome.

Results can be either the modifications made by the operation to its operands or elements created by the operation. The specification of the result describes the properties of those modifications and elements. The result specification is the only information needed (or available) about the ''service'' provided by a particular consequence. The operation's specification must therefore be complete in detailing the results it achieves for each consequence.

The *outcome* might be seen as a unifying combination of the standard and exceptional exits commonly used in exception handling, but this is not quite the case. Outcomes are fundamentally different from exceptions, both in their definition and in their usage. Existing programming languages make the assumption that exceptions must be handled after they are raised, and that the normal order of execution is sequential.

The consequence model makes no such assumptions; the outcomes of an operation can be managed in any way deemed appropriate by the designer. Since there is no split between normal and exceptional domains, there are no cases where this split is inappropriate, as there are in exception handling.

While exceptions require handling and then the termination of the frame they are raised in, a consequence may be managed by:

(1) a default continuation for that consequence (in exception handling terms, a default handler),
(2) a continuation to a next operation in the source program (sequential control flow),
(3) a user-defined ''handler'' for that consequence,
(4) mapping that consequence explicitly to the next sequential statement in the source program (performing a null operation as a handler), or
(5) promoting the consequence to form one of the consequences specified for the containing operation.

These responses can of course also be achieved through programming using Ada exceptions. By enclosing each section of a subprogram in its own frame and handling certain exceptions while propagating others, Ada's termination model can simulate the connections between the consequences of operations. But this is exactly the type of ad hoc solution that can lead to the inclusion of anomalies within a program—each individual situation requires an individually tailored new section of source program to provide the semantics desired.

The appropriate pattern for dealing with an operation's consequences should be chosen by the designer and programmer and not be enforced by the language itself. This does not mean that every consequence of every operation must be explicitly connected to a continuation—ample defaults and shortcuts are desirable to ease the programmer's burden. When the full connectivity of the consequence model is not necessary, it should not interfere with program construction. But when that connectivity is needed for an application of high dependability, it must be provided.

2. Hierarchical Program Design

Programs constructed as a hierarchy of well-defined abstractions are clearer than programs written monolithically or as ''spaghetti code''. This is because the goals of the program are compartmentalized and organized, rather than being implemented almost accidentally by a combination of patchwork and ad hoc programming. In a hierarchical program, the top-level abstraction accomplishes the goals of the program by manipulating the lower-level abstractions. Each abstraction is responsible for providing the services defined in its interface specification, and each abstraction in turn relies on the abstractions that it calls upon to provide their specified services. The most popular instantiation of these ideas is object oriented design and programming.

An oft cited reason for the inclusion of exception handling in a programming language is to enhance program modularity by extending an abstraction's responses to program states that violate its pre-conditions [14]. However, this reason is not valid for Ada due to the anomalies that exist in exception handling. Anomalies violate the assumption that an abstraction can be implemented by managing only its own concerns—abstractions in Ada may be forced to manage the concerns of lower-level abstractions also. In the example of the uncontrolled propagation anomaly, an exception propagates past the point where it can be handled appropriately within an abstraction and an upper-level abstraction must handle it, even if the propagated exception was not declared in the lower-level abstraction's interface. On the other hand, an anonymous exception cannot be handled by its name at all, and hence cannot be declared by the abstraction propagating it. In this case, all upper-level abstractions must contain an `others` exception handler to ensure that anonymous exceptions are not propagated past the point of their control. Anomalies of this kind violate the boundaries that are emplaced around an abstraction and make verification more difficult even when a program is well-structured.

Operations are defined hierarchically in the consequence model to avoid allowing a consequence to ``propagate'' past the point where it is appropriate to handle it. Either an operation handles the consequences of the lower-level operations that it invokes, or those consequences are ``inherited'' by the higher-level operation and defined as part of its specification. This restriction ensures that all relevant consequences are considered, either by managing their occurrence inside an operation or by promoting their scope to that of the operation's interface.

It might be appropriate to manage a consequence of an invoked sub-operation within the invoker, because it represents a situations for which a response can be planned. For example, an end of file consequence from a `read_character` operation is completely expected as a part of file reading. Unless the invoker is an extended form of the read operation, then there is no reason to promote the consequence to its interface.

A consequence that represents a situation common to the environment of both an abstraction and its invoked operation cannot be managed internally. This is because the consequence has greater implications than either can manage successfully. An example of this is an out of memory consequence from a dynamic data structure's `link` operation. The higher-level abstraction, perhaps a server storing requests on a list, cannot necessarily free the memory that would allow the `link` operation to succeed. In this case, the out of memory consequence needs to be promoted to the server abstraction's interface.

During the course of the design and implementation of a program, it may be possible to eliminate some consequences from consideration by limiting the operands applied to an operation. As a program stabilizes, more details are known about the interconnections between operations. Some of these details can be exploited to reduce the number of consequences concluding particular operations. For example, if an addition operation is invoked on two unknown integer operands, then it is possible that the operation will overflow because the sum of the operands is too large. If it is later realized that the operands are both of a constrained integer type that cannot generate a sum that is too large, then the overflow consequence from that addition operation can be eliminated; the partial operation of addition is made total by restricting its operands. If the program is modified, then simplifying assumptions of this sort are re-checked to guarantee that they still apply.

Real hierarchical design is made possible by the consequence model. No one can make a similar claim for the Ada exception handling model as it stands, because the anomalies cannot be prevented within that model. Restrictions must be enforced to allow abstractions to be robustly encoded in Ada. It is claimed that the restrictions necessary to allow Ada to serve as an implementation language for hierarchical programs are those restrictions that have been taken into account in the consequence model—consequences must either be managed inside an operation or promoted to its interface.

Exception handling in existing languages is exclusively programmed in a bottom-up, ad hoc manner. There are no language semantics for describing the overall exception handling goals of a program in the same way that there are for stating the standard goals. While good programmers generally design programs as a hierarchy of abstractions, the semantics for exception handling have no such abstractions. The details of the lowest level exception handler may affect the entire program in Ada, as in the uncontrolled propagation anomaly.

A model used for enhancing program dependability must allow the system designer to specify operations from the top-down. The overall policy for managing consequences (or handling exceptions) must be visible from the top-level operation of the program, rather than being composed from the interactions within the entire program (as is the case in Ada). The concerns addressed by this policy must be orthogonal to the concerns addressed by conventional control structures, and it is asserted that Ada exception handling is too intertwined with these structures to offer the power needed to ensure that a coherent policy for managing consequences is followed.

3. Verification

It is up to the designers of an abstraction to ensure that the consequence specifications for its operations are accurate. Assertions made about the result and outcome of the operation must represent the real situation within the operation. There are two paths one may follow in verifying these assertions: allow a human to be responsible for verifying the assertions, or use automated verification to prove that the assertions are accurate with respect to an operation's implementation. Depending upon the criticality of the operations being defined, one approach may be more appropriate than the other. Human endeavor is always marked by human flaws, and since automatic verification is a human activity, it may always be flawed as well. However, verification can become more and more accurate as flaws are discovered, while humans seem to improve at a less substantial rate. If utmost surety is necessary for the actual behavior of critical operations in a program, then those operations should not only be verified automatically to prove that each operation implements its interface specification, but both the operation and its verification should be inspected by humans before the operation is relied upon.

In the context of automatic verification, the assertions about an operation's consequences should either be verified or disproved. If their validity is ambiguous, then the locations of concern must be pointed out for consideration. Assuming that the specifications of lower-level operations are accurate (a fact that can be proved at a later time), then an operation can be analyzed by tracing each possible path from its operands to each of its consequences. Incorporating the assertions of each invoked operation on a particular path yields a composite statement about the characteristics of the consequence in terms of the operations on that path. This composite statement can be entered into a theorem prover to assess whether the statement of the consequence is guaranteed by the operations composing that path through the operation. If the consequence's result or outcome assertions for the operation cannot be proven, then either the operation is incorrectly implemented on that path or there is not enough information to guarantee that the assertion is true. In either case, more work is needed to ensure that the specified consequence of an operation is the actual state arrived at when the operation concludes.

Some property verifications are necessary in dependable systems. These verifications are possible only in a limited form for programs implemented in standard Ada. The consequence model directly assists in the necessary verifications because it is formal, precise, and unambiguous. The reliance that can be placed upon the verifications performed for operations based on consequences is as strong as the reliance that can be placed upon the operations composing them and upon the underlying hardware of execution (to paraphrase Hoare quoted in [15]).

SECTION 8

OZ: A Specification Notation

It is claimed that supplementing a traditional programming language with a small specification language based upon the consequence model can enhance program dependability by a very large degree. Consequences are the basis of the language, not only to avoid the pitfalls of current models based on the exception, but also because the consequence is general, allows the kind of control that designers of dependable applications need, and because it provides for the essential needs of exception handling without enforcing an inappropriate model. The initial problem of "what is wrong with exception handling in Ada?" has been generalized to "how can programs be designed to ensure that certain properties of their specifications hold?" The answer to both questions is: there needs to be a specification of exactly what operations have multiple outcomes, and exactly how these outcomes are related to the operands. The OZ language (an acronym for Operations and Zones) addresses these issues and provides a workable solution to the problems of program verification for dependability.

1. OZ Description

The specification notation as it is currently conceived has three major constructs: the *element*, the *operation*, and the *zone*. The element contains static data, the operation manipulates elements, and the zone structures concurrent operations. Each construct is defined hierarchically according to a precise format. New members of elements, operations and zones can be composed recursively from pre-existing members.

Elements:

Elements contain data in a variety of formats, many of which will be provided by the underlying language or computer of implementation. Elements can be composed from other elements, and can be passed through a filtering operation to extract one part of the element. Unlike the objects of object-oriented programming, elements have no methods associated with them; without operations they are utterly static, although they may have certain static characteristics specified in their definition.

An element's characteristics may include domain specific knowledge or assertions about the element type. For example, an integer element type might define MAXINT to be the largest representable integer. The MAXINT name can then be used to define operations on integers that do not require their specifications to be modified when the underlying machine is changed; only the MAXINT value associated with the element needs to be modified to allow the operation's specification to be used on that machine.

Another example element is a communications port. The port may represent different pieces of port status information as a corresponding bit pattern held in a segment of the port's logical pins. The element definition for the port would describe these patterns and associate a name with them to allow operations to be independent of the actual bit patterns. This allows information regarding the port element to be described formally, which in turn can simplify the operations using the element.

Operations:

Operations take elements as their operands. Application of an operation to its operands yields one consequence as its conclusion. Each consequence is composed of an outcome and result where the result is the data that flows on a particular outcome path. The outcome of an operation may be connected to another operation, called a continuation, when the result matches one or more of the operands of the continuation. Operations are thus strongly typed, but flexible in that elements can be composed and decomposed at will.

As an example, an integer addition operation is shown in Figure 4. This operation might have two operands (a and b) and two consequences. The first consequence's outcome might be called ''sum'' and the second might be called ''overflow''. The result for the sum outcome is the sum of the two operands (a+b), while the overflow outcome might have a Nil result, meaning that no actual result is produced. The *characteristic predicate* of the sum outcome that relates the outcome to the equivalence class causing it might be $a+b \leq MAXINT$, where MAXINT is the largest representable integer. The characteristic predicate of the overflow outcome is then $a+b > MAXINT$. These two predicates partition the operand space (the Cartesian product of all pairs of representable integers) into two equivalence classes.

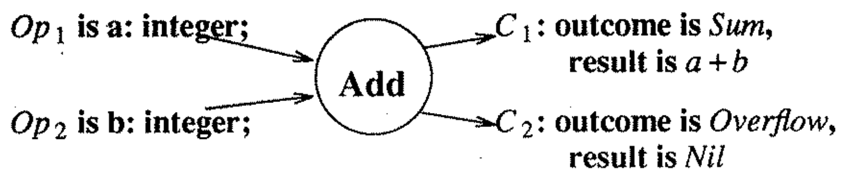
Op_1 is a: integer;
Op_2 is b: integer;
Add
C_1: outcome is *Sum*, result is $a+b$
C_2: outcome is *Overflow*, result is *Nil*

Figure 4: An Example Operation

Zones:

The zone construct is used to specify concurrent programs within the consequence model. Each zone has a boundary around it to ensure that no zone can affect another zone's computation other than through the message passing mechanism. The boundary prohibits a variable from being used by more than one zone at a time, and this removes the need for synchronizing access to the variable. Should a common data source be needed, a zone can be used to grant or deny access to the data source.

A zone accepts a set of operand messages and can generate a set of consequence messages. Each operand message is a request for a particular service from the zone and it is connected to a single operation that processes it. The service requests are queued until the zone is free to work on them—although zones are concurrent with each other, activity within a particular zone is sequential (excluding its sub-zones). Since operations are arbitrarily large, a single operation for each service request is sufficient because the operation can decode the message, compute a result and send a reply message. Consequences of the operation processing a message may: 1) become the consequence messages of the zone, 2) be deferred to a sub-zone for processing, or 3) not generate any consequence message at all.

An example zone is shown in Figure 5. This is a simple zone that controls access to a system's real time clock. Two services are provided; one for setting the clock and the other for reading the time. Communication delay may of course render the time reading slightly out of date, but this clock is not intended for fine granularity computations.

Zones ease concurrent program construction by following the same model as operations, and provide some analogous properties in the multi-processing environment.

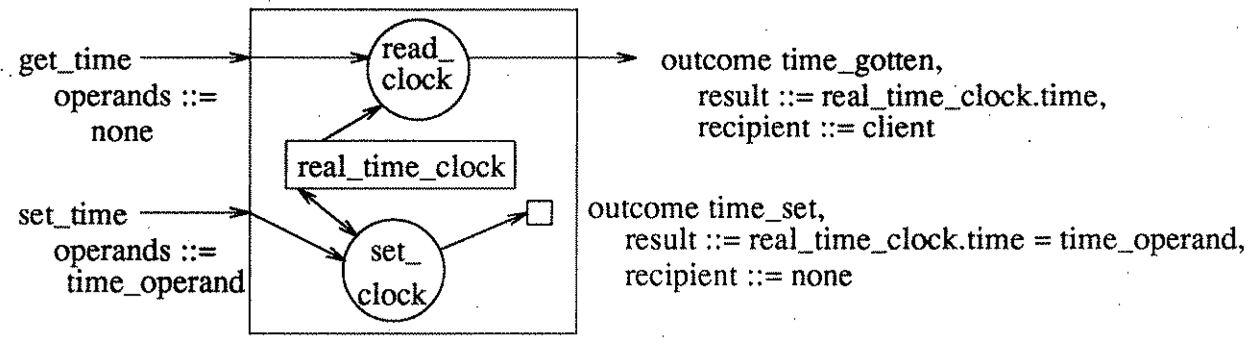

Figure 5: An Example Zone

The properties of a zone result from the properties of the operations comprising it, and there is a separation between aspects of the zone communication and aspects of the zone computation that is felt necessary to effectively reason about concurrent operations.

Overview:

The OZ notation offers a different solution to the needs of dependable computation that are currently addressed (or not) by exception handling. Because OZ is based on the consequence model, each operation is a total function over its operand space. OZ also provides semantics for constructing concurrent programs that follow in a straight-forward manner from non-concurrent operations. Both the sequentially executed operation and the concurrently executed zone are amenable to property verification, where each level of a design can be independently proven to meet its specification given that the specifications of the invoked operations are accurate. If the specifications are verified to the level of the underlying computer, then the full verification of a program's consequences is possible and its behavior is ensured to conform to its overall specification.

2. Toolset

To assist designers in using OZ to specify programs, a set of tools is necessary. These tools are not intended to provide an entire design and run-time environment, but are used to add OZ specifications to Ada programs and then verify properties of the programs. Instead of relying on Ada's flawed exception handling model, a designer can specify an operation's consequences using OZ and then implement the algorithmic portions of the operations in Ada. The OZ specifications are embedded in the Ada interface specifications for the packages and subprograms making up the design, and a full program implementation is constructed by putting together the Ada algorithms with the basic structures that implement the properties specified in OZ notation. The toolset makes this process easier and free of common errors. The toolset includes:

Specification Checker:

There are a number of properties that must hold for the definition of an operation, or that operation cannot be called "well-specified" within the model. These properties have effective algorithms for detecting whether the operation is well-specified or not, and each necessary property is checked before allowing the full Ada implementation of the program to be constructed.

Visualizer:

A data-flow graph can be constructed from operation specifications. The visualizer produces this graph and displays it on an appropriate graphic display device. The visible data-flow graph is useful for examining the connections and interactions within a design, and for obtaining an overview of a design's structure. Zooming in and out of operations will be supported to allow the entire design to be traversed and inspected. The outermost view depicts an entire non-concurrent program as a single operation, while the lower-level operations contained within it can be individually zoomed into and inspected.

Verifier:

The verifier attempts to prove that the consequences specified for an upper-level operation are actually implemented by the operations composing it. Using existing proof techniques, the verifier shows that:

(1) the operand space is partitioned by the specification for the operation's outcomes,

(2) the operation does not violate its specification by modifying inappropriate operands,

(3) the properties of each consequence are guaranteed by the specifications of the operations used to implement that consequence,

(4) zones do not observe or modify elements unless they are passed to it in an operand message,

(5) zones that are specified as infinitely operating servers are implemented as such, and

(6) communication between zones does not degrade due to badly specified message dependencies.

This set of verifications is almost certainly not complete, but will be expanded as necessary. In cases where an intended verification fails, the location of the failure will be brought to the designer's attention if such a location can be determined. At the very least, the designer will be alerted that a particular property cannot be verified for the operation or zone on which the verification was attempted.

Pre-processor:

The role of the pre-processor is to process a design that consists of OZ specifications and Ada algorithms and create from the hybrid program a pure Ada program that guarantees the specifications. This complete program can then be compiled and executed, and the properties that have been verified for the design will be honored within the executable version.

SECTION 9

OZ Examples

In order to partially justify the claims that OZ represents a workable solution to the problems of Ada, this section shows that several anomalies can be avoided by using OZ. To justify that OZ serves as a program verification technique that can be applied to Ada, an example program is presented and certain properties of it are verified.

1. Anonymous Exceptions

In OZ, there is no such thing as an anonymous consequence. Any operation invoking another as a sub-operation either connects a continuation operation to a sub-operation's consequence, or the containing operation ''inherits'' the consequence as one of its own. In the first case, the consequence is managed within the operation and has no external visibility. In the second case, the consequence has become part of the operation's external interface and is visible to any other operation that invokes it. It is thus impossible for a consequence's name to be lost.

2. Uncontrolled Propagation

OZ does not suffer from uncontrolled propagation. An operation containing a dangling, unconnected consequence internally is not well-specified according to the OZ definition, and can be detected by the checker tool. The consequences of a lower-level operation may indeed be promoted to become consequences of higher-level operations that contain it, but there is no chance of that promoted consequence being unknown to the designer of a higher-level abstraction. That consequence is detailed in the operation's interface specification, and hence must be accounted for in the higher-level operation.

3. Tasking Deadlock

The possibility of an Ada program becoming deadlocked due to delay of exception propagation has been noted above. In OZ, this situation can never arise because of the message passing paradigm employed by the Zone. Depending upon the communication mechanism, messages may be lost, but no message can cause a Zone to deadlock with another zone. There is no concept of waiting for the arrival of a particular message at all; messages are processed as they arrive, and the zone re-enters the ''message awaiting'' state after processing of the previous request is complete. If a zone wishes to implement a request and reaction to that request, then one of its operations must generate the request (and conclude), while another of its operations accepts the reply and reacts to it.

If messages can be lost by the communication mechanism, then a total deterioration of communication between the zones could ensue and activity within the concurrent OZ program would halt. A reliable communications sub-system would of course preclude this situation, but even a non-reliable message system can be used if support for watchdog timers is included. The timer could cause retransmission of a message if no acknowledgement for it is received, where the lack of acknowledgement might be a consequence of the message send operation.

4. Using OZ to Enhance Dependability in Existing Ada Programs

It is difficult to prove properties of dependability for large Ada programs because of the possible presence of exception handling anomalies in the program implementation. The presence of an anomaly is difficult to detect without a full analysis of the Ada program, but by adding a small number of OZ specifications to the program, the absence of certain anomalies can be guaranteed. The full Ada program implementation is created by the OZ pre-processor in such a way that these anomalies are provably absent in the regions that are specified as OZ operations. This section describes the general procedure for applying OZ specifications to an existing application.

Figure 6 shows the structure formed by an existing Ada program that has been designed hierarchically. A top-level control program (depicted as level 0) invokes operations on the abstractions (levels 1 and greater) to accomplish the program goals. Each abstraction is responsible for managing the data it encapsulates and for controlling the abstractions below it. However, an abstraction might not fulfill this responsibility due to the presence of an anomaly.

It is possible for an Ada abstraction to raise exceptions other than those declared in its interface due to an anomaly. Since these exception have not been declared, it is impossible to provide appropriate handlers for them without examining the abstraction's implementation—a situation that violates the abstraction's logical boundary. However, if this boundary is not violated, then the actual behavior of the abstraction within an Ada program cannot be determined. Further, even if this inspection is performed, it is very difficult to ensure that all of the possible exceptions that the abstraction can raise are discovered.

Bulletproof Abstractions:

OZ offers a solution to the problem of anomalies within abstractions without requiring a large amount of extra programming. By adding OZ specifications to the upper levels of the program, the desired behavior of abstractions can be enforced regardless of anomalies contained within lower levels. It is submitted that this solution is much more clear and dependable than a solution relying exclusively on Ada control structures, because the appropriate control structures are included by the OZ pre-

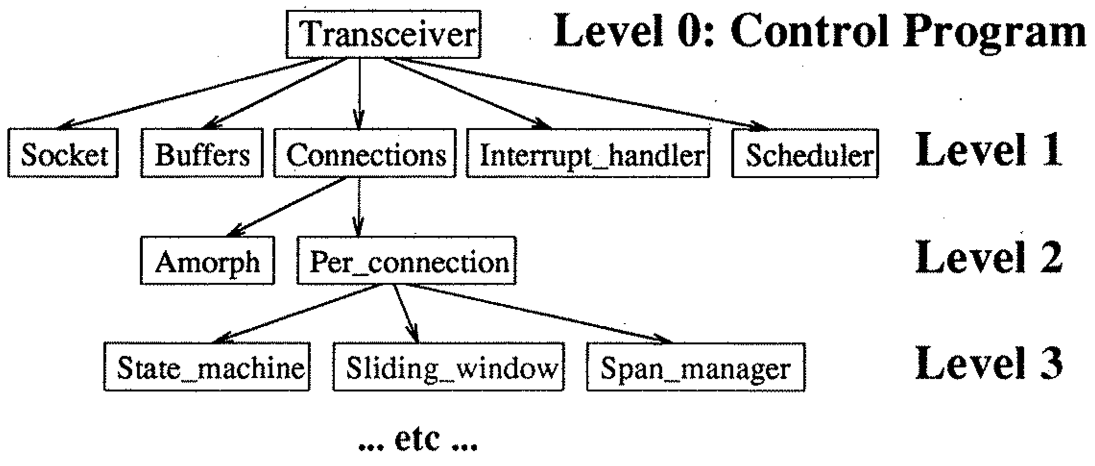

Figure 6: Hierarchical Ada Program

processor rather than being hand-coded every time they are needed.

Most of the former algorithms in the Ada subprograms can remain the same. The changes needed to add OZ specifications to the program are:

- the interface declarations of the abstraction's subprograms are stated in OZ notation as operations,
- return statements in the abstraction's subprograms are specified as representing a particular consequence of the operation,
- exception raises and handlers in the abstraction's subprograms are annotated as consequences of the operation, or are removed entirely.

These changes are sufficient to allow automatic transformation of the mixed Ada and OZ implementation into a pure Ada operation that implements the specification. The algorithmic code is surrounded with an ''anomaly-proof'' shell that enforces the consequence behavior described in the specification. Once a program region is specified as OZ operations, the synthesized program is guaranteed to conform to its consequence specifications in the following respects:

(1) an operation does not propagate any exception to its invoker,
(2) an operation concludes with one and only one of its specified consequences, and
(3) any exception raised within an operation or raised by any invoked subprogram causes the operation to conclude with the consequence

specified as the *failure_consequence*.

The first guarantee eliminates the need to handle exceptions that are propagated from the operation—there simply are none. This is sufficient to eliminate both the anomalies of anonymous exceptions and uncontrolled propagation from crossing the boundary between an OZ operation and its invoker. Instead of performing exception handling, the connections between the consequences and their continuations are specified.

The second guarantee means that invokers of OZ operations can rely on the fact that an operation will conclude as specified. It is possible that the programmer might omit one of the consequences that concludes the operation, but the pre-processor will detect this and report it before constructing the final implementation. Although total verification of the specified *results* is not yet possible, the specified *outcomes* can be relied upon as the only way that the operation concludes.

The third guarantee reflects the possibility that Ada subprograms can be completed by an exception. The Ada-only portions of the program cannot be relied upon to not raise exceptions, but the impact of those exceptions on the operation can be mitigated. The failure consequence is a necessary addition to the OZ semantics that allows OZ operations to be successfully integrated with Ada. This consequence concludes the operation whenever an exception is raised within the operation; instead of handling or propagating the exception (or both), any raised exception is uniformly treated as a failure of that operation to satisfy its specification. Unlike Clu's failure exception, the failure consequence must be managed by a continuation [6] and it is not allowed to propagate in an uncontrolled manner.

The consequence that is specified as the *failure_consequence* should be the one guaranteeing the least functionality. If no existing consequence seems an appropriate choice, a new consequence can be added for the failure. All anomalous behaviors for exception propagation are mapped into the single failure consequence. This failure can then be planned for intelligently, whereas an uncontrolled propagation does not yield so adroitly to handling—it is better to be informed of a failure in an abstraction than for the entire control program to complete before desired.

Persistent Control Programs:

Once the top-level abstractions have been transformed into operations, any propagation anomalies in the lower-level abstractions have only a well-defined representation as the failure consequence. They can no longer affect the control program in any other way. This is a potent benefit of including the OZ specifications in the program, because no analysis is required to show that these anomalies are not present—their absence is guaranteed for every version of the program that the pre-processor creates.

The control program can also be specified in OZ as concluding with several consequences. Once these consequences are specified, it can be guaranteed that the program ends with one and only one of these. The control program could further be specified as having an indefinitely long lifetime, meaning that it must not complete prematurely if it is to meet its specification. The appropriate control structures for guaranteeing this type of behavior can be automatically included by the OZ pre-processor to ensure the persistence of the program.

Applying OZ Judiciously:

The abstractions at level 2 and downwards can also be specified as OZ operations after the top-level operations have had their consequences specified. This ensures that the interactions between the top-level abstractions (level 1) and those invoked by the top-level (level 2) are well-defined, and that no anomaly can affect the relationship between them in an unexpected way. This process can be followed as far down into the hierarchy as desired. Each new level that is specified as OZ operations increases the assurance that the propagation anomalies, among others, cannot affect the vital interactions between the program's abstractions.

But wherever OZ operations are added to the program hierarchy, their presence eliminates particular anomalies before the program goals are adversely affected. The OZ operation emplaces a boundary in the program over which no exception may be propagated, and the specified consequences are guaranteed to be the only way in which the operation completes. Because these properties can be relied upon, the implementation of the program is simplified and can in turn be relied upon for certain of its characteristics.

Most of the necessary exception handling behavior for a program is encapsulated within the OZ consequence specification, instead of within ad hoc handlers. The overall dependability of the program has been increased because the OZ pre-processor checks certain properties of the specifications and inserts only dependable components with known properties into the program before allowing full compilation. Now, instead of the hit-or-miss solution that is re-attempted every time exception handling control structures are necessary in a program, the OZ-specified operations are guaranteed to be free of particular anomalies by design.

5. An Example OZ Specification & the Resulting Ada Program

To show that an OZ specification for a program can be used to define essential exception handling behavior, an example is presented that will be proven to conform to important aspects of its specification. Since operations are composed from sub-operations, verification involves proving the relationship between an operation's interface and its implementation. The interface is just the consequences of applying the operation: the equivalence classes in the operand space that cause each outcome and the characteristics of the result for that outcome are specified. The implementation consists of the sub-operations that compose the operation as well as the connections between the sub-operations. In the case of the simple example to be presented, the entire implementation is stated in terms of OZ operations; this makes it especially easy to verify the desired properties.

Due to the recursive definition for operations, total verification of an operation's consequences requires that the relationship between each interface and its implementation be proved from the top-level interface down to the underlying language or computer. This is essentially a proof of each level of the design. To demonstrate the proof process but not dwell on the details of each level, this demonstration is restricted to proving the relationship between one operation's interface and its implementation.

The specification for a *Safety Toaster* is the subject for property verification. While toasters do not commonly incorporate control programs or integrated circuits upon which to run them, it is interesting to consider how safety properties might be implemented for a particular toaster configuration. The simplicity of this example allows the safety properties to be stated in terms of the operations that interface with the toaster's sensor devices. In general, isolating safety properties is not this easy.

The underlying assumptions for this example are that the sensor devices are sufficiently reliable to provide the assurance desired and that the heat generated inside the toaster does not adversely affect the control program. One stringent requirement dominates the construction of the actual sensors; either the sensor reports accurate readings for the toaster, or it fails by producing a value that it could not possibly have measured within the toaster. The Safety Toaster control program relies upon the sensor devices to carry out its objectives.

A temperature sensor is used for the dual goals of determining the temperature at which the bread is being cooked and detecting if the bread inside the toaster is on fire. A somewhat redundant luminosity sensor is also available to the control program for reporting the light intensity inside the toaster. The light sensor can be used to detect if bread has been inserted in the toaster, and also whether the bread has caught on fire. Knowing if bread has actually been placed in the toaster allows the control program to avoid unnecessary heating.

An example of a sensor reading operation is shown in Figure 7 for the `Read_brightness` operation. One consequence (`brightness_read`) reports the brightness measured by the luminosity sensor. The other three report important information about the state of affairs in the toaster—either that the toaster is on fire, that there is no bread in the toaster, or that the luminosity sensor has failed.

The safety properties of the toaster are of the most concern here. It is desired that power to the toaster be turned off if the bread catches on fire. To make this goal possible, it is necessary that the toaster power relay be under software control, rather than under exclusively manual switch control as it is in most toasters. This ensures that if the toaster's push switch jams in the 'down' position, the power to the toaster can still be turned off.

```
--> operation read_brightness
--> consequence 1: Toast_Missing occurs when (no_toast_threshold <
-->   Light_meter) and (Light_meter < fire_threshold), result ::= Nil
--> consequence 2: Sensor_Failure occurs when (Light_meter
-->   > highest_brightness) or (Light_meter < Lowest_brightness),
-->   result ::= Nil, failure_outcome
--> consequence 3: On_Fire occurs when (fire_threshold < Light_meter) and
-->   (Light_meter < highest_brightness), result ::= Nil
--> consequence 4: Toast_present occurs when (lowest_brightness <
-->   Light_meter) and (Light_meter < no_toast_threshold), result ::=
-->   Natural(Light_meter)
function read_brightness(Light_meter : special_port) return Natural;
```

Figure 7: Read_brightness Operation

A top-level specification for the toaster control program is shown in Figure 8. This is a combination of the OZ specification for the root operation with an Ada specification for the procedure implementing the OZ specification.

The implementation of the toaster control program is shown in Figure 9, again as a mixture of OZ and Ada. This will be automatically transformed by the OZ pre-processor into the full Ada implementation shown in Figure 10. The two major operations in the program are the `Toasting` and `Safety_check` operations. The `Toasting` operation carries out the functional aims of the toaster by applying heat to the toast until it has been in the toaster as long as the table of cooking times specifies. The `Safety_check` operation ensures that the toaster does not continue cooking the toast once the toast has caught on fire. `Safety_check` will be verified to conform to the following safety properties:

```
--> operation toaster
--> consequence 1: Toast_Missing occurs when
-->   read_brightness -> Toast_Missing, result ::= yellow_alert
--> consequence 2: Sensor_Failure occurs when any -> Sensor_Failure,
-->   result ::= orange_alert, failure_outcome
--> consequence 3: On_Fire occurs when any -> On_Fire,
-->   result ::= red_alert
--> consequence 4: Done occurs when not (Toast_Missing or
-->   Sensor_Failure or On_Fire), result ::= condition_green
procedure toaster;
```

Figure 8: Ada & OZ Toaster Specification

```
procedure toaster is
  timer : Natural;
  Light_meter : special_port;
  Temp : special_port;
  Knob : special_port;
begin
  Init;
  Toast_safely(timer, Temp, Light_meter, Knob);
  --> Toast_safely -> On_Fire
  set_red_alert;
  --> Toast_safely -> Sensor_Failure
  set_orange_alert;
  --> Toast_safely -> Toast_Missing
  set_yellow_alert;
  --> Toast_safely -> Done
  set_condition_green;
end toaster;
```

Figure 9: Ada & OZ Toaster Implementation

```
with toast_safely; use toast_safely;

procedure toaster(c : out toaster_consequence) is
  failure : exception;
begin
  c.outcome := sensor_failure;
  declare
    timer : Natural;
    Light_meter : special_port;
    Temp : special_port;
    Knob : special_port;
    consequence1 : toast_safely_consequence;
  begin
    Init;
    Toast_safely(consequence1, timer, Temp, Light_meter, Knob);
    case consequence1.outcome is
      when On_Fire => set_red_alert; c.outcome := on_fire; return;
      when Toast_Missing => set_yellow_alert;
        c.outcome := yellow_alert; return;
      when Done => set_condition_green; c.outcome := done; return;
    end case;
    set_orange_alert;
    raise failure;
  end;
exception
  when others => return;
end toaster;
```

Figure 10: Full Ada Toaster Implementation

(1) The toaster is turned off if there is no toast in it.
(2) The toaster is turned off when the sensors indicate a fire inside the toaster.

(3) The toaster is turned off if the sensors indicate that they have failed.

The first is not so much a safety property as an "idiot-proof" property, because it ensures that the toaster is not heated when there is no reason to do so.

The specification for `Safety_check` is shown in Figure 11. The consequences of the sensor reading operations that indicate a violation of the safety properties have been promoted to the interface of `Safety_check`. This makes them accessible to the control program and it can react to them by shutting down the toaster power. The implementation of the `Safety_check` operation is shown in Figure 12 with the additional OZ notation necessary for connecting the consequences appropriately with management logic. The pure Ada `Safety_check` implementation is shown in Figure 13 as it might be output by the pre-processor.

The safety properties for this example can be stated as theorems to be verified in this way:

(1) For all invocations of `Read_Brightness` that cause a `No_Toast` consequence, there exists an invocation of `Power_Off` within a short time span δ.

(2) For all invocations of `Read_Brightness` or `Read_Temperature` that cause an `On_Fire` consequence, there exists an invocation of `Power_off` within a short time span δ.

(3) For all invocations of `Read_Brightness`, `Read_Temperature` or `Read_knob` that cause a `Sensor_error` consequence, there exists an invocation of `Power_off` within a short time span δ.

```
--> operation safety_check
--> consequence 1: Toast_Missing occurs when
-->   read_brightness -> Toast_Missing, result ::= Nil
--> consequence 2: Sensor_Failure occurs when any -> Sensor_Failure,
-->   result ::= Nil, failure_outcome
--> consequence 3: On_Fire occurs when any -> On_Fire,
-->   result ::= Nil
--> consequence 4: Currently_Safe occurs when not (Toast_Missing or
-->   Sensor_Failure or On_Fire), result ::= Temp & Light_meter & Knob
procedure safety_check(Light_meter, Temp, Knob : in special_port);
```

Figure 11: Safety Check Specification in Ada & OZ

```
procedure safety_check (Light_meter, Temp, Knob : in special_port) is
  light_value : Natural;
  temp_value : Natural;
  knob_value : Natural;
begin
  light_value := Read_brightness(Light_meter);
  --> (Read_brightness -> Sensor_Failure) => consequence Sensor_Failure
  return;
  --> (Read_brightness -> Toast_Missing) => consequence Toast_Missing
  return;
  --> (Read_brightness -> On_Fire) => consequence On_Fire
  return;
  --> Read_brightness -> Toast_present

  temp_value := Read_temp(Temp);
  --> Read_temp -> Sensor_Failure => consequence Sensor_Failure
  return;
  --> Read_temp -> On_Fire => consequence On_Fire
  return;
  --> Read_temp -> Okay_temp

  knob_value := Read_knob(Knob);
  --> Read_knob -> Sensor_Failure => consequence Sensor_Failure
  return;
  --> Read_knob -> Knob_read => consequence Currently_Safe
  return;
end safety_check;
```

Figure 12: Safety Check Implementation in Ada & OZ

```
procedure safety_check(c : out safety_check_consequence;
                       Light_meter, Temp, Knob : in special_port) is
  failure : exception;
begin
  c.outcome := sensor_failure;
  declare
    light_value : read_brightness_consequence;
    temp_value : read_temp_consequence;
    knob_value : read_knob_consequence;
  begin
    Read_brightness(light_value, Light_meter);
    case light_value.outcome is
      when Sensor_Failure => raise failure;
      when Toast_Missing => c.outcome := toast_missing; return;
      when On_Fire => c.outcome := on_fire; return;
      when Toast_present => null;
    end case;
    c.result.light_meter := light_value.result;

    Read_temp(temp_value, Temp);
    case temp_value.outcome is
      when Sensor_Failure => raise failure;
      when On_Fire => c.outcome := On_Fire; return;
      when Okay_temp => null;
    end case;
    c.result.temp := temp_value.result;

    Read_knob(knob_value, Knob);
    case knob_value.outcome is
      when Sensor_Failure => raise failure;
      when Knob_read => null;
    end case;
    c.result.knob := knob_value.result;
  end;
exception
  when others =>
    c.outcome := sensor_failure;
    return;
end safety_check;
```

Figure 13: Full Ada Implementation of Safety Check

The proof that these properties are guaranteed by the implementation of the toaster control program begins by examining the `Safety_check` implementation. `Safety_check` is the only operation that invokes any of the sensor reading operations—it passes the sensed values to the `Toast` operation to simplify `Toast`'s implementation. Each of the consequences mentioned in the safety properties are contained in this operation, and each one is connected to the `Power_off` operation. Therefore, the safety properties are guaranteed given that the devices function properly. Again, the simplicity of this example enabled a painless proof of the properties. In a real program, the properties to be proved are significantly more complex, but they follow this general pattern for verification.

SECTION 10

Conclusion

Exception handling addresses a real need in program construction. At the same time, exception handling is insufficient to address the full set of requirements imposed by a need for program dependability. The consequence has been presented as a model that addresses these needs for the next generation of dependable systems. To assist in the construction of these systems, the OZ specification notation is offered as the vehicle for program specification and refinement of that specification into an implementation. Precise program structuring and careful consideration of all operation consequences is an important antecedent to proving that a program conforms to its specification.

References

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

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

3.: S. Yemini, An Axiomatic Treatment of Exception Handling, *Proceedings of the 9th Symposium on Principles of Programming Languages*, January 1982, 281-288.

4.: E. Best and F. Cristian, Systematic Detection of Exception Occurrences, *Science of Computer Programming*, 1981, 115-144.

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

6.: B. H. Liskov and A. Snyder, Exception Handling in CLU, *IEEE Transactions on Software Engineering SE-5*,6 (November 1979), 546-558.

7.: D. C. Luckham and W. Polak, Ada Exception Handling: An Axiomatic Approach, *ACM Transactions on Programming Languages and Systems 2*,2 (April 1980), 225-233.

8.: A. P. Black, Exception Handling : The Case Against, Tech. Rep. 82-01-02, University of Washington, Seattle, (Reprinted) May 1983.

9.: Q. Cui and J. Gannon, Data Oriented Exception Handling in Ada, in *Proceedings of the 1990 International Conference on Computer Languages*, March 12-15, 1990, 98-106.

10.: C. A. Koeritz and J. C. Knight, Semantics of Exception Handling in the Ada Language, *To be published*, 1992.

11.: C. A. Koeritz, J. C. Knight, C. Howell and G. Bundey, Anomalies Encountered in Ada Exception Handling, *To Be Published*, 1991.

12.: DIANA Interface, *VADS: Verdix Ada Development System*, June 1990.

13.: C. A. Koeritz and J. C. Knight, Control Flow in Ada Relating to Exception Handling, *To be published*, 1992. into Algol 68 and its operational semantics. An axiomatic semantic definition for the model can be found in [27]

14.: S. Yemini and D. M. Berry, A Modular Verifiable Exception-Handling Mechanism, *ACM Transacations on Programming Languages and Systems* 7,2 (April 1985), 214-243.

15.: J. H. Fetzer, Program Verification: The Very Idea, *Communications of the ACM*, 1988.

Bibliography

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

2. L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow and G. Nelson, Modula-3 Report (revised), Technical Report number 52, Digital - Systems Research Center, November 1, 1989.

3. E. S. Roberts, Implementing Exceptions in C, Technical Report number 40, Digital - Systems Research Center, March 21, 1989.

4. C. J. Antonelli and R. A. Volz, A New Exception Handling Mechanism for Ada, *Proceedings of Ada Europe 1990*, 1990, 253-266.

5. C. N. Ausnit, N. H. Cohen, J. B. Goodenough and R. S. Eanes, Exceptions - Chapter 4, in *Ada in Practice*, Springer-Verlag, 1985, 94-127.

6. I. J. Cox and N. H. Gehani, Exception Handling in Robotics, *IEEE Computer*, March 1989, 43-49.

7. A. Koenig and B. Stroustrup, Exception Handling for C++, ATT Report , ATT Bell Laboratories, Murray Hill, NJ.

8. A. P. Black, Exception Handling : The Case Against, Tech. Rep. 82-01-02, University of Washington, Seattle, (Reprinted) May 1983.

9. J. Bolot and P. Jalote, Formal Verification of Programs with Exceptions, *FTCS 19 Digest of Papers*, June 1989, 283-290. and the notions of process and exception management. In this paper, we discuss an implementation of the low-level primitives of this system and outline the strategy by which we developed our solution.

10. V. Russo, G. Johnston and R. Campbell, Process Management and Exception Handling in Multiprocessor Operating Systems Using Object-Oriented Design Techniques, *Proceedings of the 3rd Annual Conference on OOPSLA*, September 1988, 248-258.

11. L. Boi, P. Michel, M. Buis, J. M. Jantke, J. Cazin and A. Plas, Exception Handling and Error Recovery Techniques in Modular Systems - An Application to the IASURE System, *FTCS 11 Digest of Papers*, June 1981, 62-64. into Algol 68 and its operational semantics. An axiomatic semantic definition for the model can be found in [27]

12. S. Yemini and D. M. Berry, A Modular Verifiable Exception-Handling Mechanism, *ACM Transacations on Programming Languages and Systems 7*,2 (April 1985), 214-243.

13. T. P. Baker and G. A. Riccardi, Implementing Ada Exceptions, *IEEE Software 3*,5 (September 1986), 42-51.

14. D. L. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules, *Communications of the ACM 15*,12 (December 1972), 1053-1058. uction and the practicality of the techniques are discussed.

15. L. E. Moser, Data Dependency Graphs for Ada Programs, *IEEE Transactions on Software Engineering 16*,5 (May 1990), 498-509.

16. B. H. Liskov and A. Snyder, Exception Handling in CLU, *IEEE Transactions on Software Engineering SE-5*,6 (November 1979), 546-558.

17. D. C. Luckham and W. Polak, Ada Exception Handling: An Axiomatic Approach, *ACM Transactions on Programming Languages and Systems 2*,2 (April 1980), 225-233.

18. S. Yemini, An Axiomatic Treatment of Exception Handling, *Proceedings of the 9th Symposium on Principles of Programming Languages*, January 1982, 281-288.

19. M. D. MacLaren, Exception Handling in PL/I, *Proceedings of the ACM Conference on Language Design for Reliable Software*, March 1977, 101-104. to the authors. Theseoperating systems all support the notion that the primary software structuring tool for applications will be a collection of cooperating programs (processes) mapped onto a set of loosely coupled processors.

20. I. Gertner and R. L. Gordon, Experiences with Exception Handling in Distributed Systems, *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, July 1982, 144-149.

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

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

23. F. Cristian, A Rigorous Approach to Fault-Tolerant Programming, *IEEE Transactions on Software Engineering SE-11*,1 (January 1985), 23-31.

24. F. Cristian, Issues in the Design of Highly Available Computing Systems, RJ 5856 (58907), IBM Research Division, 7 October, 1987.

25. F. Cristian, Questions to ask when designing or attempting to understand a Fault-Tolerant Distributed System, RJ 6980 (66517), IBM Research Division, 24 August, 1989.

26. E. Best and F. Cristian, Systematic Detection of Exception Occurrences, *Science of Computer Programming*, 1981, 115-144.

27. R. D. Schlichting, F. Cristian and T. D. M. Purdin, Mechanisms for Failure Handling in Distributed Programming Languages, Tech. Rep. 87-13, The University of Arizona, Tucson, Arizona, June 2, 1987.

28. F. Cristian, Exception Handling, RJ 5724 (57703), IBM Research Division, 9 July, 1987.

29. F. Cristian, Robust Data Types, *Acta Informatica 17*(1982), 365-397.

30. F. Cristian, Understanding Fault-Tolerant Distributed Systems, *Comm. ACM*, February 1991.

31. D. C. Luckham and F. W. von Henke, An Overview of Anna, a Specification Language for Ada, *IEEE Software*, March 1985, 9-22.

32. F. Cristian, Understanding Fault-Tolerant Distributed Systems, RJ 6980 (66517) , IBM Research Division, 24 August 1989.

33. P. Kruchten, Error Handling in Large, Object-Based Ada Systems, *Ada Letters X*,7 (September/October 1990), 91-103.

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

35. R. J. Abbot, Resourceful Systems for Fault Tolerance, Reliability, and Safety, *ACM Computing Surveys 22*,1 (March 1990), 35-67, ACM.

36. V. R. Basili and B. T. Perricone, Software Errors and Complexity: An Empirical Investigation, *Communications of the ACM 27*,1 (January, 1984), 42-52, ACM.

37. V. R. Basili and H. D. Rombach, Tailoring the Software Process to Project Goals and Environments, *International Conference on Software Engineering*, Monterey, CA, March 30-April 2, 1987. %P unknown.

38. G. Bernot, M. Bidoit and C. Choppy, Abstract Data Types With Exception Handling: an Initial Approach Based on a Distinction between Exceptions and Errors, *Theoretical Computer Science (Netherlands) 46*,1 (1986), 13-45.

39. D. L. Black and al., The Mach Exception Handling Facility, *ACM SIGPlan Notices 24*,1 (January 1989), 45-56.

40. M. D. Broido, Exception Handling Improves Realtime System Performance, *Computer Design 21*,11 (November 1982), 129-134.

41. G. Bull and R. Mitchell, Exception Handling Considered Harmful, in *Proceedings 1985 IFAC Real Time Programming Conference*, 1985, 93-96.

42. R. H. Campbell and B. Randell, Error Recovery in Asynchronous Systems, *IEEE Transactions on Software Engineering SE-12*,8 (August, 1986), 811-826, IEEE.

43. J. Ciesinger, A Bibliography of Error-Handling, *ACM SIGPLAN Notices Notices 14*,1 (January 1979), 16-26.

44. E. M. Clarke, E. A. Emerson and A. P. Sistla, Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, *ACM Transactions on Programming Languages and Systems 8*,2 (April 1986), 244-263.

45. N. Cocco and S. Dulli, A Mechanism for Exception Handling and its Verification Rules, *Computer Languages (UK) 7*(1982), 89-102.

46. G. F. Corliss, Design of an Ada Library of Elementary Functions with Error Handling, *Journal of Pascal, Ada, and Modula-2 6*,3 (May-June 1987), 17-31.

47. Q. Cui and J. Gannon, Data Oriented Exception Handling in Ada, in *Proceedings of the 1990 International Conference on Computer Languages*, March 12-15, 1990, 98-106.

48. C. Dony, An Object-oriented exception Handling System for an Object-Oriented Language, in *Proceedings of ECOOP 88, European Conference on Object-Oriented Programming*, S. Gjessing and al. (editors), Springer-Verlag, Berlin, August 1988, 146-161.

49. K. Efe, A Proposed Solution to the Problem of Levels in Error-Message Generation, *Comm. ACM 30*,11 (November 1987), 948-955.

50. G. Engels, U. Pletat and H. Ehrich, An Operational Semantics for Specifications of Abstract Data Types with Error Handling, *Acta Informatica (Germany) 19*,3 (1983), 235-253. grams using exception handling facilities more than Black suggests.

51. M. M. Fokkinga, Exception Handling Constructs Considered Unecessary, NR INF-84-8, Twente University of Technology, Enschede, April 1984.

52. L. D. Fosdick and L. J. Osterweil, Data Flow Analysis in Software Reliability, *Computing Surveys 8*,3 (September, 1976), 305-330, ACM.

53. N. Gibbs and al., Measures for Error Handling Effectiveness, in *2nd Annual Phoenix Conference on Computers and Communications*, IEEE, New York, March 1983, 347-349.

54. M. Gogolla, On Parametric Algebraic Specifications With Clean Error Handling, in *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, H. E. al. (editor), Springer-Verlag, Berlin, 23 March 1987, 81-95 vol 1. ption of exception-handling promotes a two-step design method for constructive algebraic specifications. In a first step the specification is given with exception detection only; in a second step exception handling is superimposed.

55. I. V. Horebeek and al., An Exception Handling Method for Constructive Algebraic Specifications, *Software - Practice and Experience 18*,5 (May 1988), 443-458.

56. W. E. Howden, A General Model for Static Analysis, *Proceedings of the Sixteenth Annual Hawaii International Conference on System Sciences*, Honolulu, 1983, 163-169.

57. W. E. Howden, Validating Programs Without Specifications, *Proceedings of the ACM Software Eng. Notes '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, Key West, Florida, December 13-15, 1989, 2-9.

58. D. T. Huang and R. A. Olsson, An Exception Handling Mechanism for SR, *Computing Languages (UK) 15*,3 (1990), 163-176.

59. T. E. Hull and al., Exception Handling in Scientific Computing, *ACM Transactions on Mathematical Software 14*,3 (September 1988), 201-217.

60. R. E. Jones and D. K. Kahaner, XERROR, the SLATEC Error Handling Package, *Software - Practice and Experience 13*,3 (March 1983), 251-257.
as a generalization of the derived type and generic concepts from Ada.

61. J. L. Knudsen, Exception Handling - A Static Approach, *Software - Practice and Experience 14*,5 (May 1984), 429-449.

62. J. L. Knudsen, Better Exception Handling in Block-Structured Systems, *IEEE Software*, May, 1987, 40-49.

63. A. Koenig, An Exceptional Idea, *Journal of Object-Oriented Programming*, July/August 1990, 52-59.

64. A. Koenig and B. Stroustrup, Exception Handling for C++, *Journal of Object-Oriented Programming 3*,2 (July/August 1990), 16-33.

65. P. A. Lee, Exception Handling in C Programs, *Software - Practice and Experience 13*,5 (May 1983), 389-405.

66. N. G. Leveson, Software Safety: Why, What, and How, Tech Report 86-04, Info & Computer Science, UC Irvine, Irvine, CA, February 1986.

67. R. Levin, *Program Structures for exceptional condition handling*, Ph.D. Thesis, Carnegie Mellon University, June 1977.

68. B. H. Liskov and S. N. Zelles, An Introduction to Formal Specifications of Data Abstractions, in *Current Trends in Progarmming Methodology, Vol. I*, vol. I , R. T. yeh (editor), Prentice-Hall, 1977, 1-32.

69. L. Y. Liu and R. K. Shyamasundar, Exception Handling in RT-CDL, *Computing Languages 15*,3 (1990), 177-192. oncurrent programs. Some analyses and optimizations on these representations are also described.

70. D. L. Long and L. A. Clark, Task Interaction Graphs For Concurrency Analysis, in *Proceedings of the 11th International Conference on Software Engineering*, ACM Press, New York, May 1989, 44-52.

71. M. Moriconi, Structure Based Formal Methods for Software Engineering, A002, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, July 27, 1989. SRI Project 5912, Contract No. N00014-83-C-0300.

72. P. G. Neumann, On Hierarchical Design of Computer Systems for Critical Applications, *IEEE Transactions on Software Engineering SE-12*,9 (September, 1986), 905-920, IEEE.

73. R. Newton, Some Exception Handling Problems in Language Systems Displaying A Multi-Path Capability, *ACM SIGPLAN Notices Notices 14*,4 (April 1979), 55-63.

74. D. L. Parnas and H. Wurges, Response to Undesired Events in Software Systems, in *Proceedings of the Second International Conference on Software Engineering*, 1976, 437-446.

75. J. A. Perkins and R. S. Gorzela, Programming Paradigms Involving Exceptions: A Software Quality Approach, *Proceedings of the Joint Ada Conference: Fifth Annual National Conference on Ada Technology and Fourth Washington Ada Symposium*, March 16-19, 1987.

76. D. E. Perry and W. M. Evangelist, An Empirical Study of Software Interface Faults, *International Symposium on New Directions in Computing*, Trondheim, Norway, August 12-14, 1985, 32-38.

77. D. Perry, Software Interconnection Models, in *Proceedings of 9th International Conference On Software Engineering*, IEEE, March 1987, 61-69.

78. D. E. Perry and M. Evangelist, An Empirical Study of Software Interface Faults - An Update, *Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences*, Kona, Hawaii, January, 1987, 113-126.

79. D. E. Perry, The Inscape Environment, in *Proceedings of the 11th International Conference on Software Engineering, Pittsburgh, PA*, IEEE, May 1989, 2-12.

80. D. E. Perry, Software Development Environments for Large Scale Software Development, in *Handout from ACM Seminar at UMD*, 20 April 1990.
PS-algol.

81. P. C. Philbrow and M. P. Atkinson, Events and Exception Handling in PS-Algol, *Computing Journal (UK) 33*,2 (April 1990), 108-125.

82. A. Podgurski and L. A. Clarke, *The Implications of Program Dependences for Software Testing, Debugging and Maintenance*, ACM, 1989.

83. A. Podgurski and L. A. Clarke, A Formal Model Of Program Dependences and its Implications for Software testing, Debugging, and Maintenance, *IEEE Transactions on Software Engineering 16*,9 (September 1990), 965-979.

84. A. Poigne, Error Handling for Parameterized Data Types, in *3rd Workshop on Theory and Applications of Abstract Data Types*, H. J. Kreowski (editor), Springer-Verlag, Berlin, November 1985, 224-239.

85. R. Prieto-Diaz and J. M. Neighbors, Module Interconnection Languages: A Survey, Tech. Rep.-189, Dept. of Information and Computer Science, UC Irvine, Irvine CA, August 1982.

86. G. Ramalingam and T. Reps, Semantics of Program Representation Graphs, 900, Computer Sciences Department, University of Wisconsin-Madison, Madison, Wisconsin, December, 1989. irically and predict their "on the average" behavior.

87. B. G. Ryder, W. Landi and H. D. Pande, Profiling an Incremental Data Flow Analysis Algorithm, *IEEE Transactions on Software Engineering 16*,2 (February 1990), 129-140.

88. V. Y. Shen, T. Yu, S. M. Tehbaut and L. R. Paulsen, Identifying Error-Prone Software - An Empirical Study, *IEEE Transactions on Software Engineering SE-11*,4 (April, 1985), 317-323, IEEE.

89. M. Spivey, A Functional Theory of Exceptions, *Science of Computer Programming 14*(June 1990), 25-42, North-Holland.

90. R. E. Strom and S. Yemini, Typestate: A Programming Language Concept for Enhancing Software Reliability, *IEEE Transactions On Software Engineering SE-12*,1 (January 1986), 157-171.

91. G. S. Tseytin, An Exception Handling Proposal for ALGOL 68, *The ALGOL Bulletin 52*(August 1988), 14-26.

92. L. L. Werner and W. E. Howden, Fault Detection in COBOL Programs by Means of Data Usage Analysis, CS87-111, Department of Computer Science and Engineering, University of California, San Diego, La Jolla, California 92093-0114, December, 1987.

93. J. M. Wing, A Study of 12 Specifications of the Library Problem, *IEEE Software Magazine*, July 1988, 66-76.

94. W. A. Wulf, R. L. London and M. Shaw, An Introduction to the Construction and Verification of Alphard Programs, *IEEE Transactions on Software Engineering SE-2*,No. 4 (December 1976), 253-265, IEEE.

95. S. Yemini and D. M. Berry, An Axiomatic Treatment of Exception Handling in an Expression-Oriented Language, *ACM Transactions on Programming Languages and Systems* 9,3 (July 1987), 391-407. ion techniques.

96. M. Young and R. N. Taylor, Rethinking the Taxonomy of Fault Detection Techniques, in *Proceedings of the 11th International Conference on Software Engineering*, ACM Press, New York, May 1989, 53-62. stem for exceptions that does not require augmented specifications. The reason for constructing such a description is not to ease proof of programs using exceptions, but is instead to better represent the semantics of exceptions, particularly propagation.

97. T. E. Lindquist, Revisiting Axiomatic Exception Propagation, in *Preliminary Proceedings of the Third IDA Workshop on Formal Specification and Verification of Ada*, W. T. Mayfield (editor), 14-16 May 1986, 5-1 - 5-13.

98. Q. Cui and J. Gannon, Data Oriented Exception Handling, *IEEE Transactions on Software Engineering 18*,5 (May 1992), 393-401.

99. M. Bretz, Ada Exception Handling, *Informationstechnik - IT (Germany) 29*,2 (1987), 89-96.

100. N. Cocco and S. Dulli, Constructs for exception handling: examples of CLU, CHILL and Ada, *Riv. Inf. (Italy) 12*,3 (1982), 161-174.

101. D. M. Berry, R. A. Kemmerer, A. von Staa and S. Yemini, Toward modular verifiable exception handling, *Journal of Computer Languages 5*(1980), 77-101.

102. S. Yemini, Task termination and exception handling in parallel programs, preprint, IBM T.J. Watson Research Laboratory, 1983.

103. D. Preston, K. Nyberg and R. Mathis, An Investigation into the Compatibility of Ada and Formal Verfication Technology, *Proceedings of the Sixth National Conference on Ada Technology*, March 1988.

104. J. M. Adamo, Exception Handling in the C/sub -/NET Parallel Programming Language, *Transputer Research and Applications 2. NATUG-2 Proceedings of the North American Transputer Users Group*, 1990.

105. L. J. Maddox and E. E. Wald, STARS Foundation Project, *STARS Information Pamphlet*, 1987.

106. C. A. Koeritz, J. C. Knight, C. Howell and G. Bundey, Anomalies Encountered in Ada Exception Handling, *To Be Published*, 1991.

107. C. A. Koeritz and J. C. Knight, An Axiomatic Basis for Understanding Exception Handling in Ada, *To Be Published*, 1991.

108. T. W. Pratt, Programming Languages: Design and Implementation.

109. J. G. P. Barnes, *Programming in Ada*, Addison-Wesley Publishing Company, 1984.

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

111. B. W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley Publishing Company, 1989.

112. C. A. Koeritz and J. C. Knight, Control Flow in Ada Relating to Exception Handling, *To be published*, 1992.

113. C. A. Koeritz and J. C. Knight, Semantics of Exception Handling in the Ada Language, *To be published*, 1992.

114. C. A. Koeritz and J. C. Knight, A Taxonomy for Problems in Ada Exception Handling, *To be published*, 1992.

115. C. A. Koeritz and J. C. Knight, The Fundamental Nature of Exceptions, *To be published*, 1992.

116. J. Banatre and V. Issarny, Exception Handling in Communicating Sequential Processes: Design, Verification and Implementation, *To appear as an INRIA Research Report*, 1992.

117. DIANA Interface, *VADS: Verdix Ada Development System*, June 1990.