Pergamon

0096-0551(94)00008-0

Comput. Lang. Vol. 32, No. 8, pp. 69–87, 1994
Copyright © 1994 Elsevier Science Ltd
Printed in Great Britain. All rights reserved
0096-0551/94 $7.00 + 0.00

EXCEPTION HANDLING: EXPECTING THE UNEXPECTED

STEVEN J. DREW and K. JOHN GOUGH

Programming Languages and Systems Group, Computer Science Department, Queensland University of Technology, Queensland, Australia 4000

(Received 4 February 1994; revision received 25 May 1994)

Abstract—Since the mid-1970s, and with the development of each new programming paradigm there has been an increasing interest in exceptions and the benefits of exception handling. With the move towards programming for ever more complex architectures, understanding basic facilities such as exception handling as an aid to improving program reliability, robustness and comprehensibility has become much more important. Interest has sparked the production of many papers both theoretical and practical, each giving a view of exceptions and exception handling from a different standpoint.

In an effort to provide a means of classifying exception handling models which may be encountered, a taxonomy is presented in this paper. As the taxonomy is developed some of the concepts of exception handling are introduced and discussed. The taxonomy is applied to a number of exception handling models in some contemporary programming languages and some observations and conclusions offered.

exceptions exception handling programming languages

1. INTRODUCTION

This paper introduces the concepts relating to exception handling, offering a working definition for an "exception" and explaining the terminology and mechanics of "exception handling" in the context of sequential computer languages and monoprocessor systems. The variety of views regarding exceptions and their handling has generated an equal variety of exception handling models. With this in mind a taxonomy for exception handler designs is developed. There are four distinguishing aspects of exception handling models which are considered in the taxonomy. The first includes the factors which affect the appearance of the exception handling model when viewing program code. Next is the method by which an exception handler is associated with a guarded region of code. The third aspect covers how an exception handling model represents and differentiates exceptional occurrences internally. The last aspect is the range of exception handler responses that the model provides. Handler responses describe the various ways in which the flow of control may be directed after execution of exception handling code.

A comparison of exception handler designs from several contemporary procedural and object-oriented languages is made and each is described in terms of the taxonomy developed. Being able to effectively classify an exception handling model gives the ability to analyse its effect upon such program characteristics as robustness, reliability and comprehensibility. Each of the exception handling models classified is discussed with respect to those characteristics. Discussion also ranges to some of the models' unique design ideas before offering some conclusions and a comprehensive list of references.

2. THE EXCEPTION

Exceptions are extraordinary events *detected* during the execution of a program. Programming errors are one cause of exceptions, in many cases however they are caused through other infrequent, unexpected but otherwise correct events. As a consequence of an exception a program may be caused to terminate precipitously. A more sinister consequence however, is that execution might continue, apparently terminating correctly, but generating erroneous results.

The part of the computer system where an exception is detected is an *exception source*. Exception sources include the hardware, operating system, language runtime support and the application program. Exceptions detected at the hardware level appear as "hardware exceptions" or interrupts.

These may be due to events detected by the processor such as the issue of an illegal instruction or an attempted division by zero. An exception detected by the operating system may appear as a signal. The Unix(r) operating system issues signals for detected events such as a "ctrl-c" keyboard interrupt and incorrect use of system memory. A language's runtime support software may perform checks on things such as values assigned to variables of bounded types or subranges. If a value is detected as being out of bounds or exceeding a specified range then this exception is expressed through the generation of a software trap. A programmer may specify tests, the failure of which might be considered an exception for a particular part of an application.

Programs are often designed to operate within a particular execution environment. A communications software package for example, may require that spurious noise in the transmission medium be kept below a certain level to ensure its correct operation. Any change in the operating environment, such as a transient burst of noise in the communication example, may have unpredictable results. In this manner the execution environment is another *exception source.*

3. EXCEPTION HANDLING

An exception handling system [1–3] specifies the program behaviour after an exception has been detected. This specification is achieved in two parts: the first being to supply instructions for remedial action required as a consequence of the exception, the second being to direct the flow of control after the instructions have been completed. Conceptually, when considering programs which utilise exception handling, two sets of code and two modes of execution can be identified. Program code can be separated into two distinct parts, code for normal execution and code for exceptional execution.

Exceptional execution is that which occurs after an exception has been detected. Code for exceptional execution is referred to as "*exception handler*" code. Normal execution is defined as that execution which is not devoted to handling an exception. Exception handler code is "associated" with a particular region of code for normal execution called the "*guarded region*". The size of the guarded region is defined by the exception handling model and might range from a single statement, to a function, to a module or an entire program. In most exception handling models the occurrence of an exception whilst executing within a guarded region, causes control to pass to the entry of its associated handler code. An exception handler may have separate sections of code each selected to be executed dependent upon the detection of a particular exception. Alternatively the handler may prescribe the same action for any exception which is detected.

4. A TAXONOMY FOR EXCEPTION HANDLER DESIGNS

There are several areas of exception handler model design into which variation may be introduced. A taxonomy is presented here which separates the important features which distinguish one exception handling model from another. The general features of interest are the *appearance* of the model, the method of *association* of exception handlers and guarded regions, the definition and representation of an *exception* and the possibilities for flow of control which are introduced through execution of an exception handler. The latter feature is often referred to as the *exception handling semantics.*

4.1. Appearance

This section deals with features of an exception handling model which are discernible from inspection of program code. It is usually apparent whether the exception handling model is an *inherent or embedded* part of the language with its own syntax, or a feature *added on* through library calls. The *granularity* of the guarded region might range from a (possibly compound) statement, a code block, a procedure or function body through to a module or whole program. Also affecting the code structure is the effective *separation* of exceptional and normal execution code. A mechanism with a clear separation will be easier to read and understand, highlighting the main purpose and extent of the guarded region and the remedial code in the exception handler section.

4.2. Association

Dynamic association of guarded regions and exception handlers is achieved when the association of a particular handler is dependent upon the path that execution has taken. This form of association is exemplified by the Unix(r) operating system signal handling facilities. In this model the signal handler code to be executed on the occurrence of a particular signal is specified through a parameter to the signal( ) library function call.

Should the programmer wish to install a different handler for a particular signal at any point in the execution, then the signal( ) function is called again with an appropriately changed parameter.

A code example below demonstrates the use and appearance of a typical though fictitious model which uses dynamic association of handlers and guarded regions. The language is some form of C and the function sethandler( ) associates the handler code designated by the parameter (handlerfuncN), with the code following the call to sethandler( ). In effect, any exception occurring in the "normal C code" sections in the example code below, will precipitate the execution of the function which was the parameter to the last call to sethandler( ). In the example, an exception occurring in the section marked /*B- -more normal C code*/ will cause handlerfunc1 ( ) to be executed. In the same manner, an exception occurring in the section marked /*C- -more normal C code*/ will cause handlerfunc3( ) to be executed. Depending upon the particulars of a model the handler might remain associated until either the function such as func1 ( ) returns, or until sethandler( ) is called again.

```
void func1 (void)
{
  /*A- -normal C code*/
  sethandler(handlerfunc1);
  /*B- -more normal C code*/
  sethandler(handlerfunc3);
  /*C- -more normal C code*/
}
void func2(void)
{
  /*D- -normal C code*/
  func1;
  sethandler(handlerfunc2);
  /*E- -more normal C code*/
}
void handlerfunc1(void)
{
  /*C code for particular remedial action*/
}
... etc.
```

Exception handler code is often associated with the guarded region through a reserved word like "rescue" (Eiffel) or "exception" (Ada). In this way the relationship between the guarded region and handler code is highlighted. This form of association is generally referred to as *static association*. An example of a code block with a statically associated handler appears in the code fragment below.

```
BEGIN
  (*Code for normal execution*)
  (*The guarded region*)
EXCEPTION
  (*Code for exceptional execution*)
  (*Exception handling code executed when an *)
```

```
  (*exception is detected whilst executing in the *)
  (*guarded region above*)
END
```

Another type of association is association through the dynamic chain and applies to the code of any procedure in an invocation chain which is initiated within a guarded region. If an exception occurs whilst executing any procedure along the invocation chain, it is the handler code associated with that guarded region which is executed. It can be seen that if the body of an invoked procedure body contained, or was itself a guarded region with an associated handler, then this second guarded region (and handler) would be dynamically enclosed within the first. In this way entry to a guarded region's code marks the entry to a new exception handling "*context*". This means that it is the handler code associated with the most recently entered (and not exited) guarded region to which control will pass on occurrence of an exception. Successful execution of the guarded region results in a normal procedure return and with it a return to the previous exception handling context.

Figure 1 depicts an invocation chain with some procedures (A, C and D) being guarded by handlers (Aex, Cex and Dex) and one procedure (B) which does not have its own handler code. If an exception occurred whilst executing any of the procedures A, C or D then control would pass to the handler code directly associated with it. This would be Aex, Cex or Dex respectively. An exception occurring during the execution of procedure B however, would cause control to pass to the handler Aex. This is because execution is currently within the exception handling context of procedure A.

Figure 2 demonstrates the dynamic nesting of guarded regions (and exception handling contexts) generated by the same invocation chain. Consider the possibility that an exception might occur whilst executing the handler code associated with procedure D (Dex). If it was desirable for an attempt to be made to handle such an exception then passing control again to Dex would be unsuitable. Clearly a loop condition may arise if the same exception recurs. Under these conditions exception handler code must be executed in the surrounding exception handling context. Thus the exception occurring whilst executing Dex would cause control to pass directly to the handler associated with the immediately enclosing guarded region, i.e. Cex.

4.3. Exceptions

Once an exception has occurred, an exception handling model may need to have some internal representation which distinguishes the particular exception. This is useful if there are particular actions which can be taken in the event of a specific exception occurring. The representation of an exception is another distinguishing feature of exception handling mechanisms. The exception may be represented as a *named entity* or simply as a value of some *variable type*. The representation

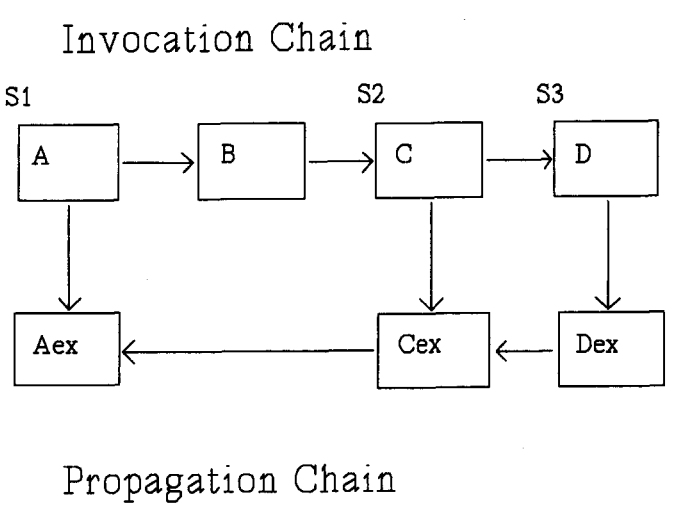

Fig. 1. A dynamic chain.

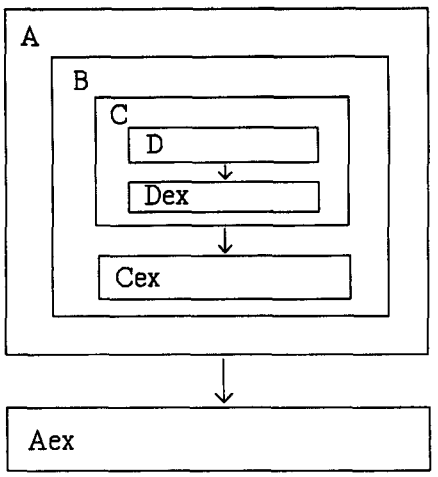

Fig. 2. Guarded regions.

may be for a "family" of exceptions, for example "maths_exception". In this case extra information about the particular exception, e.g. underflow, overflow etc. may be communicated through *parameters* associated with the exception's representation. In object-oriented languages such as C++, an exception may be represented as an *object*. In this case extra information about the exception may be contained within the description of the object, including the description of any exceptions which may be derived from that object. An example might be as above, deriving "overflow" from "maths_exception".

Some exception handling models are limited in the range of exceptional events which can be represented. This may be a result of the designer trying to keep the model platform and/or environment independent. In any case this may limit the range of *exception sources* such that *asynchronous events* (environmental events) or *signals* or user defined exceptions are not included in the set of addressable exceptional events.

Several exception handling models allow the programmer to define and explicitly "raise" an exception should a particular set of circumstances warrant for a particular application and abstraction. The above are variously termed *user, programmer or developer defined exceptions.* "*Raising*" a user defined exception is usually accomplished using a system or library function call and effectively places execution into the exceptional mode.

4.4. Execution Modes and Fail-safety

At this juncture it is useful to consider again the notions of normal and exceptional execution modes which were introduced earlier. Depending upon the implementation of an exception handling model, these notional modes of execution may have an explicit representation through some variable or data structure. If the model has no explicit representation of the execution mode, then the exceptional mode of execution may be defined as that execution existing between the start and finish of exception handler code. This extends to the execution of any dynamic chain initiated within the handler code. In the case of explicit representation, the execution mode is defined by the value of that variable or data structure. A possible advantage of some explicit representation of the execution mode is that if the execution is terminated whilst in the exceptional execution mode, this occurrence may be conveyed to the user. Such an ability is necessary to ensure program fail-safety.

A fail-safe [4–6] system is one in which every function guarantees that it will return normally only if it has executed without the detection of an exception. If the response to any exception is termination with an appropriate message then the system may be termed fail-safe. Otherwise, once having entered the exceptional mode of execution a fail-safe program is prevented from entering the normal mode unless the code of the guarded region is successfully reexecuted.

Fail-safety is related to other program properties such as robustness [7], reliability [8, 9] and is an indication of the partial correctness of a program. Flaviu Cristian [6] explains that a partially

correct program is one for which every input either generates a confined failure or termination in a state satisfying the program specification. Such partially correct programs are termed "safe" as they can never output an erroneous result. The program is said to be "fail-safe" if in a timely manner, it delivers a correct result or some other predefined output which is considered "safe" for that particular application.

4.5. Semantics of Exception Handling

Once the exception has been caught and handler code executed, there are a number of directions in which the flow of control might possibly proceed. The direction(s) allowed to the flow of control define(s) the exception handling model's *handler response semantics.*

For the purposes of further discussion, the statement being executed when an exception is detected is the "signalling statement". The block of code (or other construct) containing the signalling statement may be termed the "signaller". Yemini and Berry [10] identified five possible handler responses, designated as resume the signaller, terminate the signaller, retry the signaller, propagate the exception and transfer control. Other sources describe refinements to and variations of the above responses, often with different names, all however have semantics which might be described as subcategories of "*termination*" or "*resumption*".

Figure 3 shows some possible scenarios for flow of control with different exception handler response semantics. Both "normal A" and "normal B" are guarded regions, with associated exception handlers "handler A" and "handler B". Normal mode of execution starts in "normal A" and moves to "normal B" when B is invoked. At some point whilst executing in "normal B" an exception is detected and (in most cases) flow of control passes to "handler B" via the arrow marked "Raise". For further discussion, the guarded region "normal B" may be regarded as the signaller. The possible handler responses are indicated by the arrows leaving the bottom of "handler B".

4.5.1. Resumption

A definition of resumption semantics is given in Roy Levin's thesis [3]. With these semantics, there are two possible courses of action. The first is that execution of the handler is deferred until the signaller has completed its execution. The exception is thus "serviced" and execution resumed until a more "convenient" juncture for execution of the handler. This course might seem prudent where the signaller's action is by necessity atomic, as in the transaction on a shared database for example.

The second course, and the one most commonly regarded as resumption semantics, is that execution of the signaller is suspended until after execution of the handler. The signaller is then

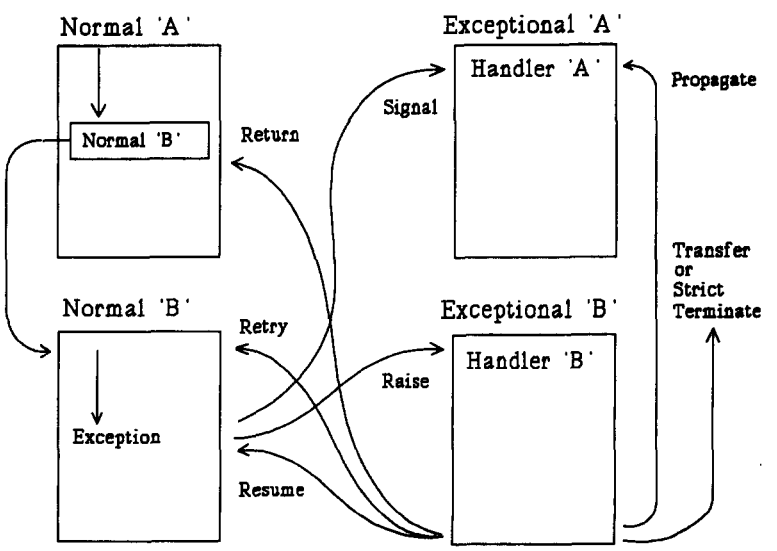

Fig. 3. Exception handler responses.

resumed immediately after the point at which the exception was detected. This latter course has been described as *Strict Resumption* [11], and it is these semantics which will be referred to as resumption in any further discussion. The corresponding flow of control is indicated by the arrow marked "Resume" in Fig. 3.

4.5.2. Termination

Termination semantics require that on the detection of an exception, the execution of the signaller be immediately terminated and control passed to the associated exception handler. After any remedial action specified by the handler code a range of handler responses are possible.

4.5.2.1. Retrial. Reexecution of the signaller after execution of the handler code is called *retrial.* In Fig. 3, the signaller is "normal B" and the flow of control for retrial semantics is indicated by the arrow marked "Retry". Retrial, in the exception handling models which support it, is usually accomplished by executing a statement such as *RETRY(optional param)*. After completing the handler code and executing the retry statement, execution is recommenced in normal mode at the beginning of "normal B". After an attempt at remedying the exception causing condition, retrial semantics allows another chance for the successful execution of the guarded region. An instance where retrial semantics might be used is in a communications program where transient "noise" corrupts data packets, requiring that they be resent after a short period of time. In some models a means of limiting the number of consecutive retries is supplied, sometimes through an optional parameter as above. The advantage of this feature is that continuous looping is avoided when for example, in the above scenario, a "transient" condition becomes unacceptably continuous.

4.5.2.2. Propagation. If an exception handling model does not support resumption and the exception handler code offers no means for retrial then execution must remain in the exceptional mode once entered. In a model where exception handling contexts are dynamically nested it may be useful for the task of handling an exception to be *propagated* to the enclosing context. Propagation is effected by signalling the currently raised exception or another from within the handler code, causing control to flow to the handler of the surrounding exception handling context. This action is indicated by the arrow marked "Propagate" in Fig. 3.

Some exception handling models support implicit propagation. In the case when the handler code is completed and the execution is still in the exceptional mode, then control automatically passes to the handler of the surrounding context. In this case the task of handling the currently raised exception is propagated. Explicit propagation is achieved by executing a signalling statement explicitly e.g. *RAISE(this_exception)* as in the case of a programmer defined exception. The net effect is the same as implicit propagation except that it is the exception represented by "this_exception" which must be handled in the surrounding context.

Nested exception handling contexts relate to different levels of abstraction in a program. As an example, "normal A" might call a number of mathematical functions as part of its operation, the code for "normal B" might perform the addition operation. As a result it could sometimes be advantageous for a different exception to be propagated which has more meaning at that level of abstraction. For instance, if an "integer overflow" exception is signalled in "normal B", and the code for "handler B" has no specific code to deal with that exception. Then a more generic "maths exception" may be signalled, for which there is specific handler code in "handler A".

Many languages including many without an exception handling mechanism, have some system for reporting the occurrence of an exception prior to relinquishing control to the operating system. This mechanism may print a message to the screen or create a memory image file for later inspection. Where exception handling is part of the language, this feature is often referred to as the "*default handler*". In a system with implicit propagation it is possible that an exception may propagate all the way out to the default handler. Where propagation must be explicit and a handler completes its code without propagating the exception, then often it is the default handler to which control will pass.

4.5.2.3. Transfer and strict termination. Some exception handling models allow control to be directed to some designated destination in the program after the execution of the handler code. The flow of control for these semantics is indicated in Fig. 3, by the arrow marked "Transfer or Strict Terminate". It is usually the case that the destination is a function from which the program cannot return and program termination follows. Such schemes are said to have *transfer* semantics.

A variation on the transfer semantics is where control is passed directly to the default handler. These semantics are called *strict termination.*

4.5.2.4. Return. The semantics described by Yemini and Berry as "terminate the signaller" [10] are essentially what we describe as *return semantics.* As indicated in Fig. 3, after execution of the handler control passes to the statement following the guarded region. From this point execution continues in the normal mode. In the case where the guarded region is a proper procedure or function than it can be seen that the semantics are exactly those of executing a return statement. Where the guarded region is a function then some models allow an alternative return value to be supplied by the handler.

4.6. Variable Scope

When considering handler response semantics and the method of association of handler and guarded region, it can be seen that the *scope of variables* whilst executing handler code is an important point. Where retrial is possible, for example, it might be desirable to alter the values of variables local to the guarded region. Under other circumstances access to "uplevel" or variables in a more global scope might also be useful.

4.7. Some Exception Handler Designs

What follows is a cursory look at various exception handler designs. The distinguishing features of each language's exception handling mechanism will be addressed as per the criteria set out in the taxonomy. Code examples are given to demonstrate the particular syntax and structure of each model. Very few languages (most notably Mesa [12] and PL/I [13] which are not covered here) offer resumption as the primary semantics for exception handling. Most modern languages support some subset of termination semantics and a few of these may include resumption semantics as an option. As a result the models chosen for the following review all support termination semantics and Exceptional C [14] demonstrates how resumption might safely be used. The review briefly covers the early exception handling models of CLU and Ada, then contrasts three object-oriented languages Modula-3, Eiffel and C++. Finally, the exception handling models designed for the C language and its derivatives are reviewed culminating in an examination of Exceptional C.

4.7.1. CLU

The CLU exception handling model [15] is an *inherent* part of the language design. The model pilots a low overhead method of setting handlers by creating tables for exception handling context information prior to runtime. Exception handlers are *statically* associated with (possibly compound) *statements.* Separation of code for normal and exceptional execution therefore, is reduced by the possibility for interleaving exception handler and normal code on a per statement basis.

```
...
begin
  ...
  Stat1 except when ohno: ExStat end
  Stat2 except
    when exc1:
        ExStat2
      exit exc1
  end
  ...
end
  except
    when ohno: ExStat2
    when exc1: ExStat4
    others: ExStat5
  end
...
```

Example: In the case of "ohno" being raised in Stat1 control passes to ExStat where it is then handled. If "ohno" is raised during ExStat or any other statement in the block containing Stat1 then it is handled by ExStat2. Similarly, if "exc1" is raised in Stat2 then ExStat2 is executed followed by the "exit" statement. The exit statement propagates the exception handling of "exc1" to the handler associated with the block which is then handled by ExStat4. Any other exceptions raised in the block containing Stat1 or while executing ExStat are handled by ExStat5. Variables in this model have the same visibility and accessibility to both exception handler code and the guarded region.

CLU exceptions are named entities which have optional parameters. Exception sources are limited to checks and traps in the language support software and programmer defined exceptions. There is no provision for dealing with "resource constraints" which indicates that only *synchronous* exceptions are handled in this model.

The mechanism can be classified as "*single-level*" *termination* which infers *explicit propagation*. Explicit propagation is achieved in two ways. The first is by using the "**exit** exception_name" statement which propagates the responsibility for handling the exception to a handler in the enclosing exception handling context but within the same procedure invocation. The other is to explicitly signal the exception, e.g. "**signal** exception_name" which terminates the procedure and raises the named exception at the procedure invocation in the caller.

Single level termination means that if there is no associated handler in that abstraction which can handle the exception, and if the task of handling the exception is not explicitly propagated before the end of handler code, then the handler automatically signals the language defined exception "*failure*". Once failure is signalled the mechanism exhibits a "multi-level" nature with implicit propagation of the failure exception. If no handler for the failure exception is found in the invocation chain then the default action is to terminate the program and to display a message string associated with the original unhandled exception. Failure may be raised explicitly though is more commonly raised automatically as it is implicit in every procedure heading.

After exception handler code has been executed, normal flow is resumed and control is passed to the statement following the executed except clause.Default use of these *return semantics* renders the exception handling model as *non-fail-safe*.

The exceptions which can be raised by the function/procedure type structure *proc*, are stated as part of the declaration or specification along with the statement of return type. This effectively gives two return paths, one through the *return* statement in the case of a normal termination, and one through the *signal* statement in the case of an exceptional termination.Static association of exceptions with invocations allows for the invoker to plan handlers for advertised exceptions, and automatically dictates that handling is performed outside of the context of the invocation or signaller.

```
prvalue=proc(i:int) returns(string) signals(ohno(string))
begin
  if condition( ) then return ("normal")
  else signal ohno ("exceptional")
  ...
```

4.7.2. Ada

The exception handling mechanism in Ada [4, 16, 17] was designed as an integral part of the language and would be classified as *inherent*. The granularity of a guarded region is the "*code block*" which starts with the word "begin" and ends with the system word "exception". The associated handler code is found between the words "exception" and "end", which marks the end of the block.Code blocks along with their associated handlers might be the bodies of procedures and nested to an arbitrary level. Variable scope is the same whether executing the guarded region or handler code. The structure imposed by the exception handling mechanism effectively separates code for normal and exceptional execution.

```
procedure proc(...) is
...
begin
  ...
  statement1;
  if error1 then raise except1; end;
  ...
exception
  when except1 = >treatment1;
  ...
end;
...
```

Ada's exceptions have *no parameters* and are *named entities* of type EXCEPTION. Ada includes four predefined synchronous exceptions which correspond to events which may be signalled by the hardware, operating system or language subsystems. These exceptions are CONSTRAINT_ERROR, signalled when a range is exceeded, NUMERIC_ERROR for operations such as an attempted divide-by-zero operation, PROGRAM_ERROR signalled when the program control structure is violated in some way, and STORAGE_ERROR which signifies the exhaustion of heap space for allocation. Possibly the only predefined asynchronous exception denotes the failure of a rendezvous operation and is named TASKING_ERROR. The programmer may also define arbitrary exception names to deal with application specific error conditions.

Ada supports the *Termination* model with *automatic* (*multilevel*) *propagation* of all unhandled exceptions.Any exceptional occurrence without an appropriate handler somewhere along the dynamic chain eventually leads to program termination and for control to pass to the operating system. If an exception is handled, i.e. appears in a when clause, the treatment prescribed is carried out and control is passed back to the dynamically enclosing block (*return semantics*) where normal execution is resumed. This action renders the exception handling model *not fail-safe*.

4.7.3. Modula-3

The Modula-3 exception handling mechanism is an *inherent* part of the language design. The granularity of the guarded region in this model is the *code block* between the reserved words "TRY" and "EXCEPT". The *TRY-EXCEPT* statement has the form:

```
PROCEDURE PROC;
BEGIN
...(*code of procedure*)
   TRY
     Body of guarded region
   (*normal execution*)
   EXCEPT
     exception1(optional_param) = >Handler1
     ||...
     ||exceptionn(optional_param) = >Handlern
   ELSE Handler0
   END;
END PROC;
```

As in Ada, the structure of the mechanism is such that there is effective separation of normal and exceptional code. The guarded region is bracketed by the reserved words "TRY" and "EXCEPT", while the associated handler (exceptionlist) falls between the words "EXCEPT" and "END". The same variables are accessible to both regions of code.

Modula-3 Exceptions are named entities and are declared in the following manner: **EXCEPTION ExId(PT)** where ExId is an exception with an *optional parameter* PT. If PT is omitted then the exception takes no parameter.

Exception sources are limited to language traps and the explicit raising of application dependent exceptions defined by the programmer. The Modula-3 Report [18] offered no means of dealing with signals or interrupts which might be generated by the operating system or hardware. This means that only *synchronous* exceptions are fielded by the model, suggesting that signals and environmental events are either ignored or cause program termination.

The PROCEDURE TYPE declaration (below) allows for the return of a value "R" which is the only return path from the procedure.

```
TYPE T=PROCEDURE(..):R RAISES {S}
```

The returned value is either that generated by the "unexceptional" execution or the value generated by an exception handler. As in the **"throw"** statement of C++, the **RAISES**{ } clause merely advertises the *set S* of exceptions that may be raised within the procedure. This feature facilitates static checks for raising of unlisted exceptions within the code and runtime checks for exceptions occurring which were not explicitly raised. If RAISES {S} is omitted then S defaults to the set of all exceptions. RAISES { } means that S is the empty set.

An exception may be named only once in the EXCEPT clause and associated ELSE part and any parameter lists are optional. The statement executes the code block between "TRY" and "EXCEPT" and if the outcome is normal then the handler code between "EXCEPT" and "END" is ignored. If a listed exception is raised then the corresponding handler code is executed. If an exception not listed is raised then the ELSE clause containing Handler0, if present, is executed. Where a handler is nominated then the outcome of the TRY statement is the outcome of the handler. This action is equivalent to *return semantics* and renders the exception handling model as *not fail-safe*. In the case of an unhandled exception, the outcome of the TRY statement is the raised exception.

The responses of the Modula-3 exception handling mechanism come under the two categories of *Termination* and *Propagation*. Propagation may be explicit or implicit by design so that either a *single or default multi-level* model may be employed. The act of handling a listed exception or the invocation of the "catch-all" (Handler0 above) option, causes the *TRY EXCEPT* statement to return the value produced by the nominated handler. The case of an unlisted exception being raised and the absence of a Handler0 clause, causes the exception to be re-raised (propagated) after the *TRY EXCEPT* statement has executed. In this case it is the unhandled exception which is propagated rather than any standard "failure" exception.

4.7.4. Eiffel

The Eiffel exception handling mechanism [4, 19] is an *inherent* part of the language. Eiffel exceptions are *typed entities* having an integer value and a string tag. The model exhibits a level of separation of exceptional and normal code similar to both Ada and Modula-3. The granularity of the model is the *procedure-like routine* described below. The guarded region itself is the "do" clause, while the exception handling code is contained in the "rescue" clause. The variables which are visible to the "do" clause have the same visibility in the "rescue" clause.

```
routine is
  require
    - -preconditions/assertions
  local
    - -variable declarations
  do
    body.
    raise(ex_name:STRING)- -raise user defined exception
  ensure
    - -postconditions/assertions
  rescue- -optional clause
    rescue_clause.
    retry
  end
```

Bertrand Meyer, the language designer, defines the cause of an exception to be the "failure" of a particular software element such that it has been unable to terminate in a normal manner. Six categories of exception are possible in Eiffel including events which are *synchronous* and *asynchronous*. The categories are violation of an executable assertion, failure of a called routine, use of an entity with a void value where an operation requires an object, impossible operation (allocation when out of memory), exceptions from the hardware or operating system (control-c hit, divide by zero) and explicit raising of (user defined) "developer exceptions" through the use of the EXCEPTIONS Kernel Library class. The EXCEPTIONS class contains the routines for querying exceptional information such as source (origin), value and cause as well as those for catching, ignoring and raising exceptions.

The handler response semantics of Eiffel are *termination* with *retrial* and *implicit propagation*. When a routine executes a component which fails and an exception results the execution is terminated and if present the rescue clause is executed (see the code sketch above). If no retry statement appears in the rescue clause then the routine returns and failure is signalled to the caller. Meyer refers to this terminate and propagate semantics as "*organised panic*". Presence of a retry statement in the rescue clause causes reexecution of the guarded region of the routine. We have described these semantics as *retrial*, Meyer refers to them as "*resumption*" presumably because successful reexecution of the guarded region is the only means of resuming the normal mode of execution.The EXCEPTIONS Kernel Library supplies an option whereby an exception can be ignored. This is helpful when the exception does not indicate a condition which is detrimental to the correct execution of the routine, e.g. a window resize. This "*ignore*" response is equivalent to what we have described as resumption semantics and Meyer terms this as a "*false alarm*". Without the possibility of misuse of the false alarm response, (if the mechanism was limited to certain signals only) the Eiffel exception handling model could be described as *fail-safe*.

4.7.5. Language C

The programming language "C" is made up of a relatively small and simple "core" language and a collection of supporting function libraries. The "C" core language does not include facilities like basic input and output, and it certainly does not include any exception handling mechanism. Any such facilities developed for "C" must be *added on* through the use of function libraries and possibly preprocessing.

The function prototypes for performing *non-local gotos* are found in the library header file *setjmp.h*. The combined actions of setjmp( ) and longjump( ) are the basis of several exception handling models developed for C and other language implementations which use C as an intermediate language. In most cases the syntactic "sugar" developed for exception handling is expanded to C code using a preprocessor. Minor additions are made to the runtime system to allow the manipulation of data structures holding context information.

Examples of the above approach for the C language are works by Lee [20] and Allman and Been [21]. Lee describes an Ada-like mechanism developed for "C" running under the Unix operating system. Exceptions are named and identified by a string value and may be programmer defined for a particular application. The model caters for both synchronous and asynchronous exceptions which appear to be treated similarly. Operating system signals (e.g. generated by control-c or an illegal address operation), are caught using the Unix signal handling facilities of "signal.h". These are then fed into the exception handling facility.

The mechanism described by Lee, like Ada's, supports *termination* and multilevel (automatic or *implicit*) *propagation* semantics, as well as an interesting non-Ada-like addition for explicitly "signalling" exceptions. The granularity of the mechanism is the code block as in Ada and the mechanism appears much the same. The same scope rules for variables applies whether executing exception handling code or guarded region.

```
NEW_EXCEPTION(AN_EXCEPTION);
NEW_EXCEPTION(SIG);
BEGIN
  fn( );
  if (condition1) exc_raise(AN_EXCEPTION);
```

```
EXCEPTION
  WHEN(AN_EXCEPTION)
    dosomething( );
  WHEN(OTHERS)
    exc_reraise;
END
```

Using the exc_raise( ) function causes a search of the associated handler for a matching "WHEN" clause in the handler. The WHEN(OTHERS) clause is a "catch-all" to handle any exception not explicitly catered for. The exc_signal( ) function does not have an equivalent in Ada and causes the search to skip the associated handler and search that of the dynamically enclosing block. The function exc_reraise( ) is used from within a handler to explicitly reraise the same exception in the surrounding context. The use of signal handling means that the model is capable of dealing with *asynchronous* exceptions. As with Ada the system would be considered *not fail-safe* as after the execution of handler code, control passes to the END statement and follows return style semantics.

Allman and Been also describe an "*add-on*" exception handling model for the "C"/Unix environment, in which exceptions are identified by a *character string*. The string contains information about the exception including its source and severity which may determine how the exception is to be handled. Three levels of severity are possible, these are Abort, which indicates that the procedure in which the exception is raised cannot continue, Error which indicates that processing can continue but the results cannot be guaranteed to be correct, and Warning which indicates that the exceptional condition has been resolved. A special severity, Transient is allocated to asynchronous exceptions (some signals) to indicate that the exception might not occur in the same place if the program was rerun. Two functions excalock( ) and excaunlock( ) are provided to enable and disable signals (interrupts) to facilitate atomic actions on data structures. Code with exception handling appears as follows:

```
a( )
{
  extern handler( );
  exchandle("?:Xname", handler);
  b( );
  excraise("W:Xname",NULL);
}
b( )
{
  excraise("E:Xname",NULL)
  ...
}
```

Exceptions handlers are set using the int exchandle(exstring, handlerproc) call, and excdhandle(..) which resets the default handler action. The first parameter is the exception string which may contain "wildcard" pattern matching characters (*,?), the second is the function name of the handler. Any parameters to the handler are gathered when the exception is raised, excraise(-exstring, arg1, arg2, ..., NULL); In this model the exception handler code does not have access to variables local to the guarded region. Variable visibility for the handler is limited to its own local variables and those in a more global scope.

The response semantics possible are *termination* which is termed "backout", *automatic propagation* and *resumption* which is termed "continue" or "ignore". The standard default handler action for an Abort severity exception is program termination, all others cause the excraise( ) to continue.The programmer determination of exception severity (rather than origin based) and the non-specific use of resumption semantics render the model as *non-fail-safe*.

4.7.6. C++

The authors Koenig and Stroustrup [22] (the source of the included code examples) outline a portable method of exception handling using C's *longjmp/setjmp* mechanism and an implementation based on that of CLU, which causes no context saving/stacking overhead. The only overhead introduced in the use of the latter exception handling mechanism is the storage of extra tables presumably for access to runtime stack information and for facilitation of the stack "unwinding" process.

Ellis and Stroustrup [23, 24] describe a model which has been accepted in the proposed ANSI C++ standard. Exceptions are *synchronous only*, corresponding to programmer defined exceptions as well as software runtime traps generated through things like range and array index checks. Operating system signals and asynchronous exceptions are not dealt with by the C++ exception handling model as they are seen as platform dependent and better dealt with through the signal handling facilities of the operating system.

The model described below is based on the fundamental idea that "a function that finds a problem that it cannot cope with, *throws* an exception, hoping that its (direct or indirect) caller can handle the problem. A function that wants to handle that kind of problem can indicate that it is willing to *catch* that exception". Exceptions in C++ are *objects* and these are the entities which are thrown. An exception is caught, however by specifying its *type*. Throwing an exception transfers control to a handler. An object is passed and the type of that object determines which handlers can catch it. When an exception is thrown, control is transferred to the nearest handler of appropriate type; "nearest" meaning the handler whose *try* block was most recently entered by the thread of control and not yet exited. The granularity of the guarded region is the *code block* as in Modula-3. Both the "try" and "catch" blocks have access to the variables which are accessible to the function (below). Any variables local to the try block are not visible to the catch statement.

```
void func(void)
(
//code of function body
try //a "try block"
{
  //...
  throw "Catch Me!";
}
catch(const char*p)
{
  //handle character string exceptions.
}
catch(...) //catch all handler.
{
  throw; //rethrow the same exception.
}
} //end of function
```

A throw expression with no operand rethrows the exception being handled. This implies that there is some repository holding the value of the "current" exception. The throw; without an operand may only appear in a catch statement or a function called by that handler. A handler declared as catch(...) is a "catch-all" handler and will perform an action for all previously unhandled exceptions of any type. The "catch-all" may only be used at the end of a block of handlers. If a thrown exception is not handled in the current exception handling context then the function terminate( ) is automatically called.

After the execution of handler code control passes back to the caller where execution resumes normal flow. The model must be regarded as *not fail-safe* on the strength of the use of these *return semantics*. The exception handling model described above, utilises *termination with explicit propagation, return and limited retrial semantics.*

It is possible to suffix a function declarator with an *exception-specification*. This is a list of the set of exceptions, and exceptions derived from these types, which *may be* thrown during a function execution. This feature accentuates the model's "single level" explicit propagation semantics.

```
void f( ) throw(X,Y)
{
  //...
}
```

Any attempt by a function to throw an exception not in its exception list will cause a call to the function "unexpected( )". As it turns out the default action for unexpected( ) is a call to "terminate( )", whose default is to call "abort( )". The functions unexpected( ) and terminate( ) execute the last functions which are passed to the functions set_unexpected( ) and set_terminate( ) respectively. These functions represent the default handling mechanism and provide a means of ensuring a "last wish" prior to program termination.

4.7.7. Exceptional C

Exceptional C [14, 25] is a superset of the C language designed to provide exception handling facilities as an *inherent* part of the language, which might normally be provided using a combination of status return values and "C" signal handling. The exception handling model uses the *termination, (single level) explicit propagation, return and resumption semantics* depending upon the exception source. Two types of exceptions are declarable. The first of these are *ordinary exceptions* which indicate program error. These are referred to above as synchronous exceptions, which are subject to termination semantics. The second are *SIGNAL* exceptions or asynchronous exceptions caused by interrupts from the keyboard or other environmental source. These are handled using resumption semantics as they represent an *environmental event* rather than a program error and as such cannot cause a reraising of the exception on resumption. The Exceptional C model supports programmer defined exceptions as well as those introduced through the runtime system traps and operating system signals.

Exceptional C handlers are associated with *code blocks* (as in Ada) and if the handler is to be associated with a statement then the statement has to be embedded within a block. The same variables are accessible to the handler code and guarded region. Exceptions may have an argument list and are declared as follows:

```
exception a, b(param-type-or-decl-list)
or
SIGNAL exception c, d(param-type-or-decl-list)
```

An interface specification must include the possible exceptions which might be raised. A function declaration for afunc( ) for example, specifies that some statement in the function might raise the exception OUT_OF_LUCK( ):

```
void*afunc(int t) raises (exception OUT_OF_LUCK(int n));
```

The occurrence of any exception other than those declared in the "raises" part of the function header causes an "ERROR" exception to be propagated to the caller. The corresponding function body would have a skeleton:

```
void*afunc(int t) raises (exception OUT_OF_LUCK(int n))
{
...
  {
    normal execution code;
  except {
    when exc1: dostuff( );
    when others: generalstuff( );
```

```
      raise OUT_OF_LUCK(3);
    when SIGNAL others: generalsignalstuff( );
  }
  }
...
}
```

Exceptions may be explicitly raised using the raise statement: raise exception_name(optional argument list); as above. This causes the block in which the exception was raised to be searched for an appropriate handler. If found the control is passed to that handler. Exception handlers are specified in the except block. This except statement is always the last statement of the code block. A handler and the block containing it may terminate if the handler completes its statement sequence or if the handler executes an *exit* or a *return* statement. Otherwise a handler may return by executing a *retry*, *next* or *resume* statement or through explicit propagation by executing a *raise* statement. The termination of a handler (and the block it is associated with) infers that flow of control passes to the statement following the block where normal execution is resumed. This being the case then the model is intrinsically *not fail-safe*.

The retry statement is retry; or retry(int); if a retry count is needed. This completes the handler code and the block in which the exception was raised is reexecuted. The next; statement returns control of execution to the statement following the block in which the exception was raised. The resume; statement, in the case of SIGNAL exceptions, causes the execution to be resumed at the point of interruption.

5. DISCUSSION AND MODEL EVALUATIONS

As introduced earlier, three important properties of programs which might be improved through an effective exception handling mechanism are those of reliability, robustness and comprehensibility. A program is reliable if it always returns results which are correct or at least safe for a given application. A program can be considered robust if it continues to operate reliably under any, even extreme or otherwise adverse operating conditions. The taxonomy offered may be used as a means of determining just how well a particular exception handling model effects these improvements. The following discussion involves the program properties above, how they relate to the classification areas of the taxonomy and the various exception handling models reviewed.

Program reliability is dependent upon the handler response semantics employed by an exception handling model. In particular, reliability is linked to the property of fail-safety, where a program produces either a correct or some other "safe" output and never an erroneous one. The only model reviewed which did not use the "unsafe" features of either return or resumption semantics, and hence the only fail-safe model, was that of the Eiffel language.

Robustness is reflected in a program's ability to overcome transient conditions which might otherwise lead to program termination. A good example would be the operation of communication software where the transmission medium is subject to "noise", and various levels of operation may need to be retried, e.g. resending a message packet after a "Nack" packet or a time-out signal is received. It is the retrial semantics in an exception handling model that provide a means of improving program robustness. The ability to provide this service over as wide a range of exception sources as possible is important. Of the reviewed models only those of Eiffel and Exceptional C provide retrial semantics and thus a means for developing more robust programs. Robustness is also promoted by providing a means for a program to cope with environmental changes. Such asynchronous changes are often signalled through remote sensors and do not indicate an error in the program. Rather, it means that the program needs to find an alternative route to its post-condition. This is best illustrated by a robot changing direction when encountering an obstacle. Any model which can field operating system signals and possesses a retrial response can effectively handle such situations. It is the Exceptional C model which most efficiently and flexibly deals with asynchronous events through the safe use of resumption semantics with the SIGNAL class exceptions.

Research [26–28] suggests that a well designed exception handling mechanism can make inroads towards the improvement of program comprehensibility in two ways. The first is the ability of the mechanism to separate code for normal and exceptional execution and the second is its ability to reduce the number of explicit conditional statements especially in the code for normal execution.

The most effective code separation technique is exemplified by the Ada model. The code for normal operation appears between the "begin" and "exception" reserved words and the exception handlers associated with that guarded region appear between "exception" and "end". Similar arrangements are used in Modula-3, Eiffel, Lee's model for "C" and also Exceptional C. The C++ model also achieves good code separation through the use of the"catch" clauses which always appear as the last clauses in the guarded region. CLU and Allman and Been's model for the "C" language, to varying extents, both allow handler code to be interspersed with normal execution code.

Logical complexity is directly proportional to the number of conditional (if ... then ... else ..., CASE, switch etc.) statements. The use of exception propagation rather than setting condition variables or conditionally returning error values obviously makes inroads into complexity reduction especially where invocations may be nested to any level. Further reduction is possible when considering the use of exception handling instead of pretesting conditions which might indicate the likelihood of a software, signal or hardware exception being raised, e.g. divide by zero, range checks etc. It follows that the number of different exceptions which can be fielded by an exception handling model gives one indication of its effectiveness in reducing complexity. All of the models reviewed could handle user defined exceptions and those defined by the language. The models of CLU, C++ and Modula-3 however, were not designed to deal with operating system signals and/or asynchronous events. This leaves the models of Exceptional C, Ada, Eiffel and the two models for "C" as being the best equipped to reduce the logical complexity of a program.

It could be argued that models which use preprocessing or have an intermediate "C" representation which is then compiled, do not actually reduce the complexity of the program as conditional statements are reintroduced after code expansion. The reader is reminded that comprehensibility is attributed to the written source coded, and it is a measure of how easy or difficult that code is to understand.

5.1. Trends and features

Table 1 provides a summary of the features of the exception handling models reviewed. From the table and inspection of other languages some interesting trends appear. Firstly with regards to appearance: nearly all of the languages reviewed had an exception handling mechanism which was part of the language design rather than providing the means for the programmer to add it on. The more modern procedural style languages tend to use the code block (statement sequence) as the granularity for the guarded region. The more modern languages also tend to provide a model which clearly separates the code for normal and exceptional execution. In the area of association, and related to appearance is the tendency towards static association of handler and guarded region. Related to the range of exceptions represented, is the tendency to provide the ability for the programmer to define exceptional circumstances for an abstraction. Not all models provide the

Table 1. Exception handler feature summary

Features/Lang.	ExcC	C++	C (A+B)	C (Lee)	Eiffel	Mod-3	Ada 8.x	CLU
Inherent part of language	Y	Y	N	N	Y	Y	Y	Y
Granularity (St'ment, Block)	B	B	B,S	B	B	B	B	B,S
Separate normal and exceptional code	Y	Y	N	Y	Y	Y	Y	N
Exceptions (Named, Typed)	N	T	T	N	N	N	N	T
Exception parameters	Y	Y	N	N	N	Y	N	Y
Association (Static, Dynamic)	S	S	D	S	S	S	S	S
Asynch. exceptions	Y	N	Y	Y	Y	N	N	N
Signals as exceptions	Y	N	Y	Y	Y	N	Y	Y
User defined exceptions	Y	Y	Y	Y	Y	Y	Y	Y
Termination semantics	Y	Y	Y	Y	Y	Y	Y	Y
Retrial	Y	N	N	N	Y	N	N	N
Propagation (Imp., Exp.)	E	E	I	I,E	I,E	I,E	I,E	I,E
Return semantics	Y	Y	Y	Y	N	Y	Y	Y
Resumption semantics	Y	N	Y	N	N	N	N	N
Fail-safe	N	N	N	N	Y	N	N	N

means for dealing with exceptions detected in the operating system or environment. This stems from the fact that the exception handling model is language based and the designer must choose between providing signal or interrupt handling for a range of platforms or none at all. The handler response semantics of choice seem to be termination with very limited use of resumption. The use of return semantics as the default form of termination response implies that the designers have a lot of faith in the programmer's ability to provide adequate alternative results through the execution of exception handler code. The limited use of retrial reflects a seemingly similar lack of understanding of the principle of fail safety and its importance.

If the question were asked: Which of the models and features reviewed stand out? The answers would be: CLU for the design of low overhead handler setting and the local reraising action of the "exit" statement. Lee's model for the "C" language with the ability to explicitly "signal" an exception, Eiffel for implementing fail-safe exception handling, and Exceptional C for the effective use of resumption semantics.

In any language which offers a non-local goto mechanism, such as the longjmp( )/setjmp( ) pair in the C language, it is possible for the programmer to construct an exception handling mechanism. Whilst possible, it proves to be an extremely difficult job to get right, which is why many languages provide a "ready made" exception handling mechanism. One requirement for the task of building an exception handling mechanism, is a detailed knowledge of the actions and side-effects of the non-local goto mechanism. For longjmp( )/setjmp( ) in C, these are well documented [29]. Another requirement is a knowledge of the desired characteristics and possible pitfalls of the exception handling model to be built. Application of the taxonomy presented to an exception handling model design, and comparison with the classifications of some existing models, should provide an invaluable aid to the designer in attaining this knowledge and also in refining design.

6. TO CONCLUDE

An introduction and overview of the notions of exceptions and exception handling have been presented and the range of attributes which may be included in exception handling model design identified. A taxonomy for exception handling models was devised with criteria for classification being the elements which affect the "appearance" of the model in the language, how guarded region and handler code are associated, the range of exceptional events and their representation in the exception handling mechanism, the types of handler response semantics that the model supports i.e. resumption, termination etc., and whether or not the model supports fail safety.

The exception handling models of eight procedural languages were reviewed with respect to the developed taxonomy. The relative merits and demerits awarded each model were discussed with emphasis on their ability to improve program robustness, reliability as well as reduce program complexity. Important design ideas were identified in the exception handling models from each of the CLU [15], Eiffel, Exceptional C [14] and Lee's model for the "C" language [20].

REFERENCES

1. Goodenough, J.B. Exception handling issues and a proposed notation. *Commun. ACM* **18**: 683–696; 1975.
2. Goodenough, J.B. Structured exception handling. *Second ACM Symp. Princ. Prog. Lang.* 204–224; 1975.
3. Levin, R.A. *Program Structures for Exceptional Condition Handling*. Ph.D. Thesis. Carnegie-Mellon University, 1977.
4. Meyer, B. *Object-Oriented Software Construction*. New York: Prentice-Hall; 1988.
5. Gough, K.J. and Holford, J. *Exception Handling with Fail-Safe Semantics and Modula-2 Implementations*. Proc. 13th Australian Comp. Sci. Conf. 127–135; 1990.
6. Cristian, F. *Exception Handling. Dependability of Resilient Computers* (Edited by Anderson T.) Oxford: BSP Professional Books. Blackwell Scientific Publications; 1989.
7. Cristian, F.Correct and Robust Programs. *IEEE Trans. Software Eng.* **SE10(2)**: 163–174; 1984.
8. Meyer, B. Writing Correct Software. *Dr. Dobb's J.* **Dec**. 48,52,54,56,58,60,63,125; 1989.
9. Buchmann, A.P. and Dayal, U. Constraint and exception handling for design reliability and maintainability. *ASME International Computers in Engineering Conference* 95–100; 1988.
10. Yemini, S. and Berry, D.M. A modular verifiable exception handling mechanism. *ACM Trans. Prog. Lang. Systems* **7**: 214–243; 1985.
11. Huang, D.T. and Olsson, R.A. An exception handling mechanism for SR. *Comput. Lang.* **15**: 163–176; 1990.
12. Mitchell, J.G., Maybury, W. and Sweet, R. *Mesa Language Manual*. Palo Alto: Xerox Research Centre; 1979.
13. *IBM OS PL/I Language Reference Manual*, SC 33-0009-2. IBM Corp.; 1973.
14. Gehani, N.H. Exceptional C or C with Exceptions. *Internal Report, AT&T Bell Laboratories*. Murray Hill, NJ 07974: AT&T Bell Laboratories; 1990.

15. Liskov, B.H. and Snyder, A. Exception handling in CLU. *IEEE Trans. Software Engng* **SE5**: 546–558; 1979.
16. US Department of Defence 1983. *Ada Language Reference Manual*. MIL-STD-1815.
17. Baker, T.P. and Riccardi, G.A. Implementing Ada Exceptions. *IEEE Software* **3**: 42–51; 1986.
18. Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow B. and Nelson, G. *Modula3 Report (revised)*. Draft of 21 October 1989.
19. Meyer, B. Applying "design by contract". *IEEE Comput.* **40**–51; 1992.
20. Lee, P.A. Exception handling in C programs. *Software Pract. Exp.* **13**: 389–405; 1983.
21. Allman, E. and Been, D. An exception handler for C. *Usenix Association Summer Conf. Proc.* 25–45; 1985.
22. Koenig, A. and Stroustrup, B. Exception handling for C++ (revised). *USENIX Proc. C++ Conf.* 149–176; 1990.
23. Ellis, M.A. and Stroustrup, B. *The Annotated C++ Reference Manual*, Chap. 15, pp. 353–367./ Reading, MA: Addison Wesley; 1990.
24. Stroustrup, B. *The C++ Programming Language*, 2nd Edn., Chap. 9, pp. 293–324. Reading, MA: Addison Wesley; 1991.
25. Cox, I.J. and Gehani, N.H. Exception handling in robotics. *Computer* **22**: 43–49; 1989.
26. Weinberg, G.M. *The Psychology of Computer Programming*, Chap. 12, pp. 217–245. New York: Van Nostrand Reinhold; 1971.
27. Shneiderman, B. *Software Psychology. Human Factors in Computer and Information Systems*, Chaps 3 and 5, pp. 38–62, 92–121. Cambridge, MA: Winthrop Publishers, Inc.; 1980.
28. Boehm-Davis, D.A., Holt, R.W. and Schultz, A.C. The role of program structure in program maintenance. *Int. J. Man-Machine Studies* **36**: 21–63; 1992.
29. Plauger, P.J. *The Standard C Library*. Chaps 8 and 9, pp. 181–204. New York: Prentice Hall; 1991.

About the Author—STEVEN J. DREW received the B.App.Sc.(Hons) degree from Queensland University of Technology in 1990, and is currently working towards his Ph.D. in the Computer Science Department of the same University. His research interests are in computer languages and graphics. Current research focuses particularly on exception handling mechanisms and the application of the "fail-safety" principle.

About the Author—K. JOHN GOUGH currently holds the position of Director of Research for the Faculty of Information Technology at Queensland University of Technology. Professor Gough also heads the Computer Languages and Systems Group where the research concentrations are in code generation technology, reverse engineering, exception handling, object-oriented languages utilizing the type extension concept, "object-oriented" operating system kernels and secure application program environments.