

# A Study of the Applicability of Existing Exception-Handling Techniques to Component-Based Real-Time Software Technology

JUN LANG and DAVID B. STEWART  
University of Maryland

---

This study focuses on the current state of error-handling technology and concludes with recommendations for further research in error handling for component-based real-time software. With real-time programs growing in size and complexity, the quality and cost of developing and maintaining them are still deep concerns to embedded software industries. Component-based software is a promising approach in reducing development cost while increasing quality and reliability. As with any other real-time software, component-based software needs exception detection and handling mechanisms to satisfy reliability requirements. The current lack of suitable error-handling techniques can make an application composed of reusable software nondeterministic and difficult to understand in the presence of errors.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.6 [**Software Engineering**]: Programming Environments; D.2.10 [**Software Engineering**]: Design; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.4.1 [**Operating Systems**]: Process Management; D.4.7 [**Operating Systems**]: Organization and Design—*real-time systems and embedded systems*; D.4.8 [**Operating Systems**]: Performance; D.4.9 [**Operating Systems**]: Systems Programs and Utilities

General Terms: Design, Languages, Performance, Standardization

Additional Key Words and Phrases: Component-based software, error detection and handling, faults, reconfigurable software, signals, survey, timing and deadline failures

---

## 1. INTRODUCTION

Component-based software is becoming increasingly popular as a means of rapidly creating real-time applications by assembling software building blocks [Bihari and Gopinath 1992; Gertz et al. 1995; Schneider et al. 1995;

---

Authors' addresses: Department of Electrical Engineering, Institute for Advanced Computing Studies, University of Maryland, College Park, MD 20742; email: {jlang; dstewart}@eng.umd.edu.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

© 1998 ACM 0164-0925/98/0300-0274 \$5.00

Stewart et al. 1992b]. The focus of developing component-based software has been on the structure of initialization and normal operation code, with little thought given to exception handling. Research into exception handling, on the other hand, has targeted errors in non-real-time systems. There has hardly been any research into exception handling for component-based real-time software (CB-RTS).

This study on exception handling is a step in our research toward creating a software framework for CB-RTS [Stewart et al. 1997]. The key contributions of this study are the following:

- Identification of the primary limitations to using existing exception-handling techniques in CB-RTS. The limitations include lack of time-bounded handler determination, no support for external handling, lack of any form of criticality management, and no suitable exception specification for operating-system-based mechanisms.
- Identification of existing exception-handling techniques, relating to exception representation, handler binding, exception raising, information passing, handler scope, and resource cleanup, that can be used in CB-RTS.
- A comprehensive survey that compares and contrasts exception-handling techniques used in programming languages and operating systems. The survey details are suitable for any software designer and are not limited to programmers of component-based real-time systems. We conclude that an exception-handling mechanism suitable for CB-RTS is likely a hybrid between a programming-language-based mechanism and an operating-system-based mechanism.

Brief background information regarding exception handling, component-based software, and real-time systems is given next, followed by the details of our study.

## 2. BACKGROUND

Authors of different exception-handling mechanisms have their own definitions of exceptions. Goodenough [1975a; 1975b] defines *exception conditions* as those brought to the attention of the operation's invoker, which becomes part of normal exit or return. Gehani [1992] defines *exception* as an error or an event that occurs unexpectedly or infrequently, which includes an error or a signal. Cui [1989] defines *exceptions* as *implementation insufficiencies*, which exclude software errors, unanticipated program conditions, domain failures, and range failures. The most commonly accepted definition of *exception*, that we use in this study, is the union of "error," "exceptional case," "rare situation," and "unusual event." The entity that is raising an exception stops and waits for the completion of the exception processing.

In real-time systems, errors are typically categorized as follows [Cox and Gehani 1989; Stewart et al. 1997]:

- Software errors* are caused by design oversights or implementation mistakes in the software. They are also called *design errors* and informally called “bugs.” Dividing by zero, attempting to access an array element through an index that is too large or too small, incorrectly implementing a mathematical equation, failure to identify a special case that needs special processing, or incorrect loop exit conditions are examples of software errors.
- Hardware errors* involve the failure of the underlying hardware. Examples are unintentional halting of a processor in a multiprocessor system, a sensor sending incorrect data, and a memory parity error.
- State errors* result from the difference between the system’s perception of the external environment and the actual environment. State errors are also called *external status errors*. For instance, a robotic manipulator is carrying a pencil, but the pencil drops. A state error occurs because the internal state says “robot has pencil” while in reality the robot is not holding the pencil.
- Timing errors* occur when operations do not satisfy the timing constraints imposed in a real-time system. Missing a deadline and using more CPU time than reserved in the worst case are examples of timing errors.

Exceptions are usually divided into two classes [Cristian 1982; Goodenough 1975a]: *predefined* and *user-defined*. *Predefined* exceptions are declared implicitly and are associated with conditions that are detected by the underlying hardware or operating system; they are also called *system-defined* exceptions. *User-defined* exceptions are defined and detected at the application level.

A *signal* is a notification of an event and may occur frequently; the signaler does not usually wait for the completion of the signal handling. Examples of signals include timer expiration and a request to kill a process. In this survey signals are not considered exceptions.

*Exception handling* is the immediate response and consequent action taken to handle the exceptions. An *exception handler* is the code attached to (or associated with) an entity for one or several exceptions and is executed when any of these exceptions occur within the entity. Depending on the exception-handling mechanism, an entity can be a program, a procedure, a statement, an expression, an object, or data. Note that fault tolerance [Pradhan 1996] is not included within the definitions of exception handling and has not been investigated by this study.

## 2.1 Exception Handling in C and UNIX

Perhaps the most common form of exception-handling method used by software programmers is the “return-code” technique that was popularized as part of C and UNIX, as shown in the following example:

```

if ((fd = open(filename, O_RDONLY)) == -1) {
    fprintf (stderr, "Error %d opening file %s\n", errno, filename);
    exit( );
}

```

Most general-purpose operating systems and real-time operating systems (RTOS) such as Solaris 2.x, VxWorks, VRTX, and QNX still use this method, especially for errors within system calls.<sup>1</sup> Although this technique is the most popular, it has many major drawbacks:

- Error Prone*: Checking return values by *if*-statements is easy for programmers to ignore. If the programmer forgets to check a function, errors may enter the system unexpectedly and show up later. For instance, how often does a program check the return value of *printf( )*? More importantly, if a layer of the software fails to check the condition, every trace of the error may be lost. This problem causes hard-to-detect bugs.
- Poor Modular Decomposition*: The main execution thread of the operation is mixed with *if*-statements and error-handling code. As a result, neither the main operation code nor the exception-handling code can be easily understood and maintained. In some cases, it is not even possible to distinguish which parts of the code are specifically for exception handling, and which are not.
- Poor Testability*: It is difficult to analytically verify that every possible error has a known handler, and it is hard to test every scenario in a systematic manner.
- Inconsistency*: The return value which denotes error is inconsistent. Some functions return NULL to indicate errors while others use  $-1$ . In some cases, the type of the return value of a function must be changed just to accommodate the error code. For example, the function *getc( )* is defined to return a value of type *int* instead of *char* just to be able to return  $-1$ .
- Lack of Information*: Any additional data crucial to handling exceptions must be passed outside the return code method. Usually global variables such as *errno* in UNIX/C must be used. This in turn complicates the design of multithreaded systems and reduces the reusability of software modules. For debugging purposes, no information where the error occurred in the code is passed to the caller.

Addressing the weaknesses of the return-code error-handling method has fueled the research into exception-handling mechanisms. An overview of these other mechanisms is given next.

### 3. OVERVIEW OF CURRENT EXCEPTION-HANDLING MECHANISMS

This study focuses only on mechanisms used at the system software level. This includes techniques embedded into both programming languages and operating systems. The study does not consider mechanisms defined at the hardware level [Thekkath and Levy 1994], nor techniques used internally by applications as a layer above the system software level [Strong and Miller 1995].

---

<sup>1</sup>Information regarding VxWorks can be obtained at <http://www.wrs.com/products/html/vxwks52.html>; regarding VRTX at <http://www.microtec.com/products/vrtx.html>; and regarding QNX at <http://www.qnx.com/product/qnxrtos.html>.

Table I. Comparison of Language-Based and Operating-System-Based Mechanisms

	Programming-language-based mechanisms	Operating-system-based mechanisms
Advantages	Good portability Good readability	Handling both system and application level exceptions Language independent exception handling
Disadvantages	Cannot handle system level exceptions Language specific handling methods	Less portable to a different operating system No static checking

We classify exception-handling mechanisms into two categories: *programming-language-based* and *operating-system-based*, as shown in Table I. A programming-language-based mechanism is the exception handling defined by a programming language and implemented by the language's compiler. An operating-system-based mechanism uses library or system calls provided by the operating system. Each mechanism has its own merits. The advantage of one mechanism is often the disadvantage of the other; thus design compromises are often necessary.

A program using a programming-language-based method can be ported to all platforms that support this language. Since programming-language-based mechanisms provide special syntax for exception handling, and many people prefer special syntax rather than system calls, programming-language-based mechanisms are often considered to have better readability than operating-system-based mechanisms.

Code that uses an operating-system-based mechanism may be less portable to a different platform; the part that is not portable is difficult to rewrite in the absence of a similar exception-handling facility. And because operating-system-based mechanisms are not integrated into the programming language, no static checking can be performed.

Programming-language-based methods typically cannot handle all exceptions uniformly [Lindsay 1977]. For example, if an error occurs in a system call or by code written in a different language, the exception cannot be caught directly by the language's facility. Therefore, the error cannot be handled in the same way as exceptions raised by the language's own mechanism. The programmer is forced to use the method provided by the operating system or other language to detect and handle the exception. The handling action is often to raise again the exception using the language's own facility.

In contrast, operating-system-based mechanisms can handle exceptions occurring in both the system and application levels and can provide consistent exception-handling methods. These handling methods are independent from programming language implementations and thus can provide compatibility among programs written in different languages.

More details of the trade-offs between programming language and operating system mechanisms are considered in Section 5.

There are too many exception-handling mechanisms proposed or implemented to describe them all in depth. Instead, we choose to study in detail a representative and influential set of exception-handling mechanisms that demonstrate the many different approaches. They are listed in Table II. At the base of this table is a list of other mechanisms that we considered, but do not study in depth. These methods are either an implementation of an

Table II. Selected Exception-Handling Mechanisms

Classification	Exception Handling Mechanism	References	Abbreviation used in this paper	Similar Mechanisms
Operating system based mechanisms	Mach exception handling facility	[Black et al. 1988]	Mach	Accent
	Global error handling mechanisms in Chimera	[Stewart et al. 1992], [Stewart 1994], [Stewart & Khosla 1997]	Chimera	
	Structured exception handling of Windows NT	[Custer 1993], [Niezgoda et al. 1993]	Windows NT	
Programming language based mechanisms	Goodenough's exception handling notation	[Goodenough 1975a], [Goodenough 1975b]	Goodenough's notation	Cocco's mechanism
	Exception handling in CLU	[Liskov & Snyder 1979]	CLU	
	Ada-83 and Ada-95 exception handling	[DoD83 1983], [Barnes 1995], [Gauthier 1995], [Tang 1992]	Ada	Lee's mechanism; Amsterdam's mechanism
	Exception handling extension to AMLX	[Nackman & Taylor 1984]	AMLX	Zetalisp
	Yemini's replacement model	[Yemini & Berry 1985]	Replacement Model	
	Real-Time Euclid	[Kligerman & Stoyenko 1986]	RT Euclid	
	Knudsen's exception handling mechanism	[Knudsen 1987]	Knudsen's mechanism	
	Dony's object-oriented exception handling	[Dony 1988], [Dony 1990]	Lore	
	C++ exception handling	[Ellis & Carroll 1995], [Koenig & Stroustrup 1990], [Stroustrup 1991]	C++	Portable C++, Better C, Java
	Modula-3 exception handling mechanism	[Cardelli et al. 1988]	Modula-3	Roberts' mechanism
Cui's Data-oriented exception handling	[Cui 1989]	Cui's mechanism		
Exceptional C	[Gehani 1992]	Exceptional C		
<p>Many other programming language and operating system exception handling mechanisms have been used. However, the methods used are all similar to one or more of the mechanisms listed above. To avoid repetition, details of these other mechanisms are not included in this paper. However, we do include below a reference to each of these other mechanisms that we considered.</p> <p>Other operating-system-based mechanisms are <i>Medusa</i> operating system [Ousterhout 1980], Lindsay's mechanism [Lindsay 1977], and <i>Accent</i> (a predecessor of Mach) operating system [Rashid &amp; Robertson 1981].</p> <p>Other programming-language-based exception handling mechanisms are <i>Java</i> [Campione &amp; Wairath 1996], Portable implementation of C++ exception handling [Cameron et al. 1992], <i>Better C</i> [Wang 1994], <i>ML</i> [Milner &amp; Tofte 1991], [Paulson 1991], <i>MESA</i> [Mitchell et al. 1979], <i>COMMON LISP</i> [Wilensky 1984], <i>ZETALISP</i> [Weinreb &amp; Moon 1981], <i>MACLISP</i> [Moon 1974], <i>INTERLISP</i> [Teitelman 1974], <i>PL/I</i> [Noble 1968], <i>SMALLTALK-80</i> [Goldberg &amp; Robson 1983], Levin's mechanism [Levin 1977], Roberts' exception implementation in C [Roberts 1989] (based on Modula-3), Lee's "Exception Handling in C programs" [Lee 1983] (based on Ada), Cocco's mechanism [Cocco &amp; Dulii 1982], Borgida's mechanism [Borgida 1986] and some real-time programming languages such as RTC [Wolfe et al. 1991], Flex [Kenny &amp; Lin 1991], [Lin &amp; Natarajan 1988] and RTC++ [Ishikawa et al. 1990].</p>				

equivalent mechanism in a different programming language or operating system, or each aspect of the mechanism is already covered by one of the other methods. For example, Portable C++ and Better C are implementations of C++'s exception handling as language extensions to C. Java's mechanism is almost the same as C++'s except that it adds the **finally** construct which is used in Modula-3 and Windows NT. Therefore, the error-handling techniques used in Java, Better C, and Portable C++ are covered by our study of C++, Modula-3, and Windows NT.

In the remainder of this section, we provide an overview of the basic elements of the selected mechanisms. More detailed descriptions of each mechanism can be found in the corresponding references, listed at the end of each paragraph. The approaches are compared and contrasted in Section 5, using the evaluation criteria that we first outline in Section 4. Sample code for some of these mechanisms is given in the appendix. Our conclusions are summarized in Section 6.

### 3.1 Programming-Language-Based Mechanisms

Goodenough's notation is the first structured exception-handling mechanism proposed. It either terminates or resumes the program's execution

after an exception is handled. As a result, Goodenough's notation declares three kinds of exceptions: **Escape**, which can only terminate; **Notify**, which can only resume; and **Signal**, which can either terminate or resume. It also uses the same three keywords to raise the corresponding exceptions. If an exception is raised from an operation, the mechanism first tries to find a local handler, which is at the same context as the operation. If it cannot find a local handler, the system searches the handlers provided by its direct caller. A handler must exist for the raised exception; otherwise the program aborts. In this notation, handlers are associated with statements. Also, users can define their own default handler for each exception and can free resources using **cleanup** handlers. Exceptions must be specified in the procedure header [Goodenough 1975a; 1975b].

CLU is based on a simple model of exception handling and can support only termination. It uses the same algorithm as Goodenough's notation to search for an exception handler. Since the mechanism searches only one level up besides the local context, it is called a single-level termination model. If a user wants to raise an exception several levels up, he must raise the same exception explicitly in the handler of each level. This exception propagation method is thus called *explicit propagation*. A construct **except . . . end** is used to attach handlers to a program statement. Exceptions are raised by using the statements **signal** or **exit**. An exception can carry parameters with it and must be declared in the heading of the procedure which might raise it [Liskov and Snyder 1979].

Ada declares exceptions by the statement **exception**. An exception is raised by using a **raise** statement. A **begin . . . exception . . . end** construct is used to bind handlers to a code block. An exception not handled is automatically raised into the upper levels along the calling chain until a handler is found or until the program boundary is reached. Therefore, this propagation method is called *automatic propagation*. Ada also supports only termination. This exception-handling mechanism can be disabled by users [Barnes 1995; Department of Defense 1983; Gauthier 1995; Tang 1992].

AML/X is a dynamically scoped programming language which allows run-time determination of an identifier's binding. By exploiting this feature it uses assignment statements, initialization, or declarations instead of specific primitives to bind a handler to an exception. Meanwhile, it can also bind another exception identifier to the exception. By doing so recursively, it can achieve hierarchical exception handling. The handler can be a procedure, an expression, a boolean, or a label. Exceptions are raised by the procedure **raise\_exception( )**. These exceptions propagate along the identifier chain instead of the calling chain. AML/X supports termination and resumption, but the handler is executed only when the operation will resume [Nackman and Taylor 1984].

The Replacement Model associates handlers with expressions. The handler's result replaces the expression's result when an exception is invoked. Its semantics of signaling an exception are very similar to that of calling a procedure. The Replacement Model has single-level exception determina-

tion and explicit propagation, but supports both termination and resumption. It depends on the compiler to check for the exceptions that are not handled; hence the exceptions must be declared in the procedure interfaces [Yemini and Berry 1985].

Real-Time Euclid is a real-time programming language that can detect timing errors. It uses **kill** and **deactivate** statements to terminate a process or to deactivate a process in the cycle when an exception is detected. It also uses an **except** statement to raise an exception that must resume. A handler can be bound to the process or procedure to catch this exception. Each handler shows its intent to catch certain exceptions by specifying them in its header [Kligerman and Stoyenko 1986].

Knudsen's mechanism uses the *sequel* concept, which is semantically similar to the *procedure* concept to declare the exception and bind the handler. A sequel is declared as **Sequel S(. . .) Begin . . . End**, where **S(. . .)** is the name of an exception combined with its arguments; the code within **Begin . . . End** is the handler associated with it. Knudsen's mechanism supports only termination; the termination level is the block in which the sequel is declared. A prefixed sequel is used to achieve hierarchical exception handling. Exception propagation is realized by passing sequels as procedure parameters [Knudsen 1987].

Dony's object-oriented exception handling defines exceptions as subclasses of the special class **ExceptionClass**. In this mechanism, handlers can be attached to exceptions, classes, or expressions. Raising exceptions is done by sending exception messages to an exception object with the statement **signal**. This mechanism supports automatic propagation and provides a primitive **when-exit** to cleanup resources when the signaler terminates (its stack is unwound). Dony's mechanism supports termination, resumption, and retry [Dony 1988; 1990].

In C++, there is no specific declaration for an exception. Users can raise an ordinary object as an exception by using the statement **throw**. A **try{. . .} catch{. . .}** structure attaches handlers led by **catch** to a guarded block of code led by **try**. If the handler for a raised exception cannot be found locally, C++ unwinds the stack of the **try** block and propagates the exception to its caller. This procedure continues until a handler is found or until the default handler is called, which then aborts the program. After a handler is found and executed the **try** block is terminated. Execution continues at the first statement after the **try** block to which the executed handler is attached. Exceptions can be specified in the function header [Ellis and Carroll 1995; Koenig and Stroustrup 1990; Stroustrup 1991].

Modula-3 uses the statement **exception** to declare an exception and the statement **raise** to raise an exception in conjunction with an argument. The constructs **try . . . except . . . end** and **try . . . finally . . . end** are used to bind handlers to a code block and to clean up resources. The code led by **finally** is executed whether or not exceptions are raised. If the code is for resource cleanup, then it is guaranteed to be executed no matter what happens. Only termination is supported in Modula-3 [Cardelli et al. 1988].

Cui's data-oriented exception handling uses Ada's exception handling to declare and raise exceptions. However, it restricts exception handlers to attach themselves to only packages (objects). Users can overload the generic exception handler declared in the object definition with their own handlers when the object is instantiated. As a result, exception propagation is not needed. Termination is the only supported handling action in this mechanism, but parameters can be passed with exceptions [Cui 1989].

In declaring and raising exceptions, Exceptional C is the same as Modula-3, but the number of arguments of an exception has no limit. Handlers are provided in the **except**{. . .} block attached to a code block. Automatic propagation is used in Exceptional C, but the exceptions across function boundaries must be specified in the function header. Otherwise, the default exception **ERROR** is propagated. Exceptional C supports termination, resumption, and retry [Gehani 1992].

### 3.2 Operating-System-Based Mechanisms

In Mach, exceptions are messages, and exception handlers are independent tasks which receive these messages as their input. An exception handler is bound to a task by registering an exception port to the task. This is completed by kernel routines such as **task\_set\_exception\_port**( ). Exceptions are raised by sending messages to this port. Exception RPCs **raise**, **wait**, **catch**, and **clear** are used to raise exceptions, to suspend execution, to receive raised exceptions, and to send handling results respectively. The registered ports are scanned to determine exception handlers, and the exception can be propagated from the thread level to the process level. Failing tasks terminate or resume after exception handling [Black et al. 1988].

Chimera is an RTOS with two separate mechanisms for handling general errors [Stewart et al. 1992a] and timing errors [Stewart and Khosla 1997]. In the mechanism for general errors, exceptions are defined and raised by the function **errInvoke**( ). Handlers are ordinary functions and are bound to exceptions by calling **errHandler**( ). The exception handler is determined by checking the handler chain. The default handler is supplied for exceptions with no specific handler defined. In the mechanism for timing errors, exceptions are detected by the RTOS. Users bind handlers to the exceptions by using **tfhInstall**( ). The priority of the handlers can be different from that of the failing task. Termination, resumption, and retry are supported in Chimera.

Windows NT's Win32 API provides programmers with compiler-independent exception-handling system calls. Consequently, Windows NT's mechanism is not language specific, and each language defines how the underlying exception-handling mechanism is exposed. Microsoft C is the first programming language to use the Windows NT mechanism. In Microsoft C, programmers use the Win32 routine **RaiseException**( ) to raise exceptions with its parameters. The construct **try**{. . .} **except**{. . .} **finally**{. . .}, which is similar to that of Modula-3, is used to bind handlers and cleanup

resources. The handler determination and propagation methods are the same as C++'s. A filter routine **GetExceptCode( )** is used to discover which exception is raised. Windows NT supports termination and resumption and provides programmers with systemwide default handlers. An important feature of Windows NT is that its exception handler can be another thread or process. This feature, however, is not supported in Microsoft C [Custer 1993; Niezgodna et al. 1993].

#### 4. EVALUATION CRITERIA

We have established the criteria that we will use to evaluate whether or not a particular exception-handling method can be used for CB-RTS. The criteria have been developed based on our extensive work in developing the Chimera Methodology, a component-based software engineering paradigm for creating reusable real-time software [Stewart 1994; Stewart et al. 1997]. Although there exist other CB-RTS approaches, most have the same exception-handling needs.

The evaluation criteria are classified into three categories: *essential requirements*, *desirable requirements*, and *performance goals*. The requirements and goals are often directly or subtly interrelated, and are sometimes contradictory. Thus any solution is likely to be a compromise between functionality and performance.

##### 4.1 Essential Functional Requirements

An essential functional requirement must be satisfied in order to provide a mechanism that is suitable for CB-RTS. If the requirement is not satisfied, then some aspect of component-based design or real-time system design is compromised. Following are the essential requirements and the effect of not satisfying each one.

*Reusability.* Software components are reusable. An exception handler that is bound to a software component must also be reusable. Alternately, exceptions and exception handlers are defined independent of the component, thus reused independently.

If reusability is not satisfied, it compromises the ability to incorporate exception handling into software components. It forces programmers to write exception-handling code even if the main body of code is already available.

*Encapsulation.* Software components have strict encapsulation requirements. All data and functions are local, unless explicitly exported. An exception-handling mechanism must also differentiate between exceptions that are detected and handled locally and those exceptions that are detected internally but exported for other components to handle. Results of exported exception handling must be importable [Dony 1988].

Encapsulation is an essential ingredient to any component-based software paradigm. If this requirement is not supported, component-based software development is compromised.

*Predictability.* A real-time system is still predictable even after an exception occurs. The overhead of raising an exception and determining the handler must be time-bounded and preferably  $O(1)$ . The exception handling of a failing task should not cause higher-priority critical tasks to miss their deadlines unless the higher-priority tasks are directly affected by the failing task.

Although just ensuring the mechanism is time bounded does not make the application predictable, not meeting this requirement makes it impossible for the application-level software to be predictable. To ensure time-bounded exception handling, user-defined exception handlers must also be time bounded. In this study, however, we only focus on the mechanisms and not the user-defined policies.

*Exception Handler Binding.* It must be possible to add a new exception handler or change the currently installed exception handler as the system is running, without shutting down or rebooting the system.

Many critical real-time systems cannot be shut down and rebooted. The exception handling, however, may be a function of the environment or the current needs of an application. If this requirement is not satisfied, a CB-RTS component might need to be rewritten for every application or environmental circumstance, just because the exception-handling requirements differ. This violates the fundamental definition of component-based software design.

*Distributed Processing.* An exception handler component can be another thread or process. It can run on another processor if it is in a distributed shared-memory or network environment. The software component raising an exception is completely independent from the component that deals with the exception.

Because each CB-RTS component runs as a separate process or thread and is transparent to the processor on which it is running, a mechanism that does not support distributed processing forces the programmer to transfer exceptions to other processes manually. Consequently there is no consistent structure for exception propagation, which affects the ability to assemble a new application from existing software components.

*Totality.* A mechanism can detect all types of errors and raise their corresponding exceptions [Lindsay 1977]. In addition to the conventional hardware errors and software errors, the mechanism must be able to detect state and timing errors and provide methods to handle them.

It is important to detect and handle state errors and timing errors in real-time systems, in such a manner that the system does not fully fail. If a mechanism does not support detection and handling of all types of errors, there is the potential for critical failure should those errors occur.

*Criticality Management.* Priority or criticality of an exception handler should be flexible and in general under control of the software. In most real-time systems, processes are priority driven, using either static or

dynamic priorities. Such priorities must be maintained, even by exception handling, in order to prevent priority inversion or error explosion.

In many real-time systems, the handling of an error is not necessarily the highest-priority event. For example, in a flight control system, it is more important to execute the control algorithms that ensures continued flight of the airplane, rather than handle an exception that is indicating that a temperature sensor is providing faulty data. An exception-handling mechanism without some form of criticality management is likely to always execute an exception handler at highest priority; such practice could prove damaging or fatal in a real-time system.

Determining the priority of an exception is a subtle design decision to avoid unnecessary priority inversion leading to timing errors. However, this is a policy decision, not a mechanism decision. As a result, the exception-handling mechanism should provide a method to control the priority of an exception, but need not to determine the priority.

*Consistency.* A task can keep its state consistent when an exception occurs. Allocated resources should be cleaned up when a task is terminated because of an exception. Because real-time systems usually continue to execute even in the presence of errors to prevent catastrophic failures, the integrity of shared data and resources must always be maintained.

A mechanism that is not compatible with the synchronization methods used to guard access to critical sections can lead to additional errors. Since the system must do its best to recover from the original errors, it is not acceptable for the exception-handling mechanism to ignore the integrity issues of the shared resources.

## 4.2 Desirable Requirements

Desirable requirements include readability, ease of use, and compatibility. They are helpful, but are not crucial to the correct operation of a mechanism. Although these requirements are subjective, there are still some basic principles that should be observed by all mechanisms.

*Readability.* Some mechanisms have a higher capacity to improve a program's readability than others. Good readability reduces errors and the cost of maintenance. However, determining readability is subjective. Often, if a mechanism can provide the same functionality as another mechanism with fewer lines of code, fewer nested operations, or can separate the exception-handling code from the main execution code, it is considered to have better readability.

Good readability is important to an exception-handling mechanism because it is one of the founding principles for structured exception handling. If a mechanism does not improve the readability, people are reluctant to use it.

*Ease of Use.* There must be as little extra work as possible for programmers using a mechanism. A mechanism that provides a set of standard

templates or a CASE tool to speed the development is often considered easier to use.

Ease of use is significant for the success of a mechanism. If a mechanism is difficult to use, no one will use it even if it provides very good functionality.

*Compatibility.* A mechanism can catch and handle the exceptions raised by external modules. It is desirable for operating-system-based mechanisms to coexist with mechanisms that are part of programming languages, and vice versa, and for mechanisms to be compatible across the different programming languages which may be used in a large application.

Mechanisms that are not compatible increase development and maintenance cost and reduce the reliability of exception handling.

### 4.3 Performance Goals

Performance goals include *reliability*, *efficiency*, *development cost*, and *response time*. If two methods provide the same functionality, the one with better performance is desired. There is always a trade-off between functionality and performance. If a design cannot meet its performance specifications while keeping all the required functionality, a designer may choose to sacrifice the functional requirements to get better performance. From our previous description of requirements in Section 4.1, the designer knows what is going to be lost if an essential requirement is not met.

*Reliability* reflects the quality and robustness of a mechanism and the program using the mechanism. It should be maximized. A mechanism itself should not introduce new errors into programs. It should have some facilities to help the programmer prevent or detect additional errors. It should also encourage programmers to think more thoroughly about exception-handling strategies, or help them write good exception handlers. This in turn increases the reliability of the program.

*Efficiency*, also known as *overhead*, reflects the resources needed to execute the mechanism, such as CPU time, memory, disk space, and communication channels. Overhead should be minimized.

*Response time* is the time that a mechanism needs from raising an exception to invoking an associated handler, and it must be optimized. Response of an exception-handling mechanism is important to real-time systems, since exceptional situations usually require quick handling; otherwise the result may be dangerous or catastrophic. Mechanisms with quick responses are desirable.

*Development cost* refers to the time and effort of the application programmer (hence labor costs) invested in designing and implementing a desired policy given an exception handler mechanism. The mechanism requiring lowest development cost while meeting the other requirements is desired.

Note that in real-time systems, performance goals are not necessarily the same as in non-real-time systems. For example, the overhead during initialization is not crucial. In most non-real-time environments, one of the main goals of an exception-handling mechanism is to minimize the perfor-

mance overhead of defining exceptions and binding handlers under normal system operation. The penalties of exception handling only occur when an exception is in fact detected. The design goal is achieved at the cost of large overhead in raising exceptions and selecting the appropriate handler for a specific exception [Koenig and Stroustrup 1990; Liskov and Snyder 1979].

Real-time systems, in contrast, have many timing constraints, and the amount of time to detect and handle an exception must be bounded. For example, if brakes need to be applied to a moving locomotive to prevent an accident, any delay in applying those brakes may result in the train not stopping in time. In most real-time systems, it is acceptable to increase the overhead of initializing a mechanism, in favor of reducing the overhead and bounding the time to detect and handle an exception when it occurs [Stankovic 1988].

## 5. APPROACH COMPARISON

Different implementation approaches of major components of the current exception-handling mechanisms are compared in this section. Following are the major components of an exception-handling mechanism that we study, which are derived from basic requirements of applications that use exceptions:

- (1) Exception representation
- (2) Handler binding
- (3) Exception raising
- (4) Handler determination
- (5) Information passing
- (6) Handler scope
- (7) Resource cleanup
- (8) Exception interface
- (9) Criticality management
- (10) Reliability checks

We now compare the approaches to realize these elements one by one. A summary of the comparison is listed in Table III.

### 5.1 Exception Representation

*Exception representation* defines what an exception is and how it is represented in an exception-handling mechanism. An exception can be defined as either a special or an ordinary data type, a data structure, a procedure, a message, or a primitive similar to a procedure. Current exception representations organize exceptions into one of the three structures: *singular*, *hierarchical*, or *object* structure.

Many of the mechanisms shown in Table III use the singular structure. The singular structure is shown in Figure 1(a). Each exception is unrelated to the others, as there is no way to group exceptions together. In the hierarchical structure, as shown in Figure 1(b), an exception can have several subexceptions, but any exception can have at most one parent.

Table III. Summary of Exception-Handling Mechanisms (see Table II for references)

Categories	Mechanism	Example (See Appendix)	5.1	5.2	5.3	5.4
			Exception Represent.	Handler Binding	Exception Raising	Handler Determination
Operating System Based Mechanisms	Mach		Singular	Dynamic	Message passing	Handler pool
	Chimera	Listing 6	Hierarchical	Dynamic	control flow	Handler pool
	Windows NT	Listing 4	Singular	Semi-dynamic	Trap	Handler pool + stack unwinding
Programming Language Based Mechanisms	Goodenough's notation	Listing 7	Singular	Semi-dynamic	control flow	Single-level stack unwinding
	CLU		Singular	Semi-dynamic	control flow	Handler pool + stack unwinding
	Ada	Listing 3	Singular	Semi-dynamic	control flow	Stack unwinding
	AML/X		Hierarchical	Dynamic	control flow	Identifier binding
	Replacement Model		Singular	Semi-dynamic	control flow	Single-level stack unwinding
	RT Euclid		Singular	Semi-dynamic	control flow	Stack unwinding
	Knudsen's mechanism		Hierarchical	Static	control flow	Identifier binding
	Lore		Object	Semi-dynamic	Message passing	Handler pool + stack unwinding
	C++	Listing 2	Object	Semi-dynamic	control flow	Stack unwinding
	Modula-3		Singular	Semi-dynamic	control flow	Stack unwinding
Cui's mechanism		Singular	Static	control flow	Instance scanning	
Exceptional C	Listing 5	Singular	Semi-dynamic	control flow	Stack unwinding	
Conclusion	Existing approaches applicable to CBRTS?		Object, Hierarchical	Dynamic Semi-dynamic	Message passing, Trap, Combination	None of above

Mechanism	5.5	5.6	5.7	5.8	5.9	5.10
	Information Passing	Handler Scope	Resource Cleanup	Exception Interface	Criticality Management	Reliability Checks
Mach	Limited	Global	None	None	None	Dynamic
Chimera	Limited	Hybrid	None	None	Timing errors only.	Dynamic
Windows NT	Limited	Local	Construct	None	None	Dynamic
Goodenough's notation	None	Local	Construct	Function header	None	Static
CLU	Complete	Local	None	Function header	None	Dynamic
Ada	Limited	Local	None	None	None	Dynamic
AML/X	Limited	Global	None	None	None	Dynamic
Replacement Model	Complete	Local	None	Function header	None	Static
RT Euclid	None	Global	Disable raising excps	None	None	Dynamic
Knudsen's mechanism	Complete	Global	Construct	Function header	None	Static
Lore	Complete	Hybrid	Construct	None	None	Dynamic
C++	Complete	Local	Construct	Function header	None	Dynamic
Modula-3	Limited	Local	Construct	None	None	Dynamic
Cui's mechanism	Complete	Global	None	Function header	None	Dynamic
Exceptional C	Complete	Local	None	Function header	None	Dynamic
Existing approaches applicable to CBRTS?	Complete, Limited	Hybrid is preferred	None are ideal.	Function header	None of above.	Static, Dynamic

AML/X, Knudsen's mechanism, and Chimera have this exception structure. Chimera's mechanism, however, is limited to only two levels. As shown in Figure 1(c), object structure relaxes the restriction on the number of parents, as compared to the hierarchical structure. Therefore, an exception can have both several children and several parents. C++ and Lore support this exception organization.

Advantages of the object exception structure are that it directly maps the native relation of exceptions so the handler for a parent exception can

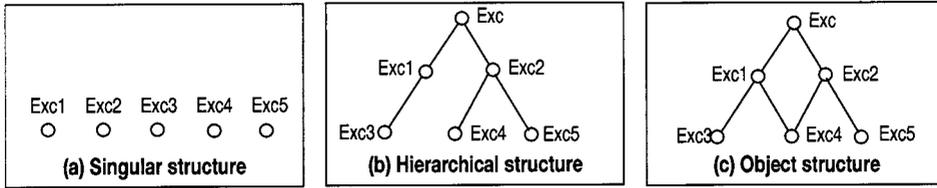


Fig. 1. Diagrammatic view of exception representations.

naturally handle all its child exceptions. Consequently, less handler bindings are needed, and the program is shorter. This improves readability and makes the mechanism easier to use. For instance, if a handler wants to handle all the exceptions *Exc*, *Exc1*, . . . , *Exc5*, binding the handler to *Exc* achieves this aim in the object structure. In the singular structure, the handler must be attached to each exception.

Because hierarchical exception organization still has restrictions compared to object structure, the natural relationship between exceptions cannot be completely realized, and the advantages mentioned above cannot be fully exploited. Consider the following scenario: When *Exc4* occurs, there are no handlers attached to either *Exc4* or *Exc2*, but there is a handler for *Exc1*. We want this handler to deal with *Exc4*. In the hierarchical structure, we need to attach the handler to *Exc4*, but in the object structure, this operation is unnecessary. Therefore, the readability and ease of use of the hierarchical structure are not as good as those of the object structure.

Any exception organization is applicable to CB-RTS. A mechanism can choose to use one of them based on the consideration of other desirable requirements such as performance overhead and implementation cost. If these desirable requirements can be satisfied, the object or hierarchical structure is preferred because of its readability and ease-of-use advantages.

## 5.2 Handler Binding

*Handler binding* attaches handlers to certain exceptions to catch their occurrences in the whole program or part of the program. It is also called *handler association*. There are three ways to bind handlers with exceptions: *static*, *semi-dynamic*, and *dynamic* binding.

In static binding, once a handler is attached to an exception, the same handler is used for every occurrence of that exception in the whole program or process. The handler is associated with the exception in different contexts of the program. Knudsen's mechanism uses this method. The exception and its handler are defined together and associated statically. Cui's data-oriented exception handling is another example of static binding. Although different instances of the same object can have different handlers, the handler binding is permanent.

Semidynamic binding associates different handlers with the exception in different contexts during a program's execution. It uses binding constructs such as **try{...} catch{...}** to attach handlers. In the context of the **try**

block, this association is still static. Most programming-language-based mechanisms support this kind of handler binding.

The dynamic binding method can attach different handlers to the exception in the same context. Which handler should be used cannot be determined at compile-time and must be checked during run-time. Dynamic binding usually uses system calls or imperatives. The setup calls for dynamic binding, however, introduce overhead to the main line execution of a program [Lindsay 1977]. This overhead should be minimized or controlled, e.g., the overhead occurs in initialization instead of run-time. Chimera, Mach, and AML/X are examples of the dynamic handler binding method.

The semidynamic binding method can be used to achieve the functionality similar to that of the dynamic method. A semidynamically bound handler can call different handler routines based on run-time conditions, except that once a handler needs to be changed the program needs to be recompiled. However, static binding cannot achieve this because the run-time condition may not be valid in some contexts.

Static binding does not satisfy the essential requirements of reusability and exception handler binding because the handler code must be rewritten in order to change a handler. Dynamic binding or semidynamic binding, however, does satisfy these requirements because it naturally allows different handlers to be used based on run-time conditions.

### 5.3 Exception Raising

*Exception raising* is the notification of an exception's occurrence. Exception detection and raising for predefined exceptions and user-defined exceptions are different. For predefined exceptions, detection and notification are usually performed implicitly by the run-time system, by either hardware or software. For user-defined exceptions, the user must explicitly test the exception conditions and raise the exceptions.

For the implementation of exception raising, current mechanisms use *control flow transfer*, *message passing*, or *trap*, which is a software interrupt, to raise an exception.

Most mechanisms notify the occurrence of exceptions by transferring the control flow. For instance, Exceptional C uses *setjmp/longjmp* routines; Chimera uses a function call *errInvoke()*; and C++ uses the **throw** statement.

Message passing is another way to raise exceptions. In Lore, raising an exception is accomplished by first creating an instance of the exception class and then sending a message to one of its methods (e.g., *look\_for\_handlers*). Mach realizes this by sending messages to exception ports registered for the task.

Windows NT uses the *trap* method. When an exception turns up, a *trap* occurs which captures an executing thread, switches it from the user mode into the kernel mode, and transfers the control to a fixed location in the operating system, which is called a *trap handler*. The trap handler notifies

a kernel module, called the *exception dispatcher*, that an exception has occurred.

While control flow transfer has the advantages of quick response and high efficiency, message passing and the *trap* method enable distributed processing, by allowing an exception to propagate beyond the boundaries of the process. The advantage of the *trap* method is that it immediately invokes the operating system's process management code. Its disadvantage is the performance penalty because the kernel must be called every time an error occurs [Niezgoda et al. 1993]. Message passing is also suitable for distributed processing. Its advantage is explicit operating-system-independent transfer of an exception. Its disadvantage is large communication overhead, with possible unbounded time delay.

Based on our comparison, message passing or the *trap* method can be applied to CB-RTS because they are essential to distributed processing. A method which combines control flow transfer with message passing or *trap* may also be applicable. The best design for exception raising is likely to be a compromise between these methods.

#### 5.4 Handler Determination

*Handler determination* is the process of receiving the notification, identifying the exception, and determining the associated handler. Methods used to determine handlers and to propagate exceptions vary greatly. They can be divided into the following categories:

- Stack unwinding
- Handler pool
- Combination of stack unwinding and handler pool
- Backtracking exception identifier bindings
- Scanning instances of objects

One of the main issues with handler determination is the time it takes to find the proper handler once an exception is raised, which we call the *worst-case search time* (WCST).

C++, Ada, RT Euclid, and Modula-3 use stack unwinding to search for an associated handler. This technique linearly checks exception handlers statically defined on the current program block (context). The handler defined first is checked first. If none can be found to handle the raised exception, the context stack is unwound, and the search begins within the new context. The order of stack unwinding is in reverse order of the actual *calling chain*, which is the sequence of nested operation calls.

Handler stack unwinding is a variant of the stack-unwinding technique. This technique maintains another stack consisting of only the stack frames guarded by the handlers. This stack is searched and unwound to find the handler. Once the handler is found and the stack frame is determined, the main stack is unwound directly to this frame, and the handler is executed. Exceptional C uses this technique.

Another variant is single-level stack unwinding. The stack unwinds only once. If a handler is still not found in the new context a fatal error occurs, and the program aborts. Compiler checking is used to assure that a handler is supplied, either locally or in the caller. Goodenough's notation and the Replacement Model use this method.

The WCST of stack unwinding is proportional to the length of the calling chain and the total number of handlers defined. Because the execution path of a program cannot be predicted before running, the length of the calling chain or the number of defined handlers are also unpredictable. As a result, the WCST is unbounded. Conversely, the WCST of its variants are proportional only to the number of all handlers or local handlers defined and are time unbounded.

A handler pool is a handler chain (i.e., a linked list) or a table of handlers, each of which has been bound to a specific exception or group of exceptions. Chimera uses a handler chain; Mach uses a port table. To find an associated handler, the pool is searched linearly. WCST is proportional to the total number of handlers defined. CLU, Lore, and Windows NT combine stack unwinding with the handler pool technique: a separate handler chain or table is stored within the stack frame. As with both stack unwinding and pure handler pool methods, the WCST of this technique is not time bounded.

AML/X backtracks exception identifier bindings to determine a matching handler. Similar techniques are used in Knudsen's mechanism. A sequel can prefix another one to achieve hierarchical handling. The sequence of the prefixed sequel should be backtracked to find the first handler to be executed. The WCST of this method is proportional to the number of the bindings. This number however is not bounded.

Cui's mechanism scans all the instances of an object for handler determination, since users can supply different handlers for the same exception raised in different instances. As a result, the WCST is proportional to the number of instances. Consequently it is unbounded.

Although handler determination methods adopted by different exception-handling mechanisms look very different, none of them are sufficiently predictable for a real-time system, since the amount of time to determine the proper exception handler is not time bounded. In theory, if a mechanism uses the singular structure, static binding, and treats the exception raised from different instances of a guarded object as the same, then its handler determination time should be time bounded. However, no current mechanisms use this design, and the design would not satisfy the essential requirements, since static binding and singular structure are not suitable for CB-RTS, as described in prior sections. Thus, no current approach satisfies the essential requirements for CB-RTS, so further research into handler determination is needed.

## 5.5 Information Passing

*Information passing* transfers information useful to the treatment of an exception from its raising context to its handlers. The information should

be passed together with the notification of the exception, and it may include the name, description, location, severity of the exception, and other useful data. Since the exception might be raised outside the current environment, the saved information should be independent from the raising environment. Current exception-handling mechanisms either support no information passing, limited information passing, or complete information passing. No user-defined information can be passed to the exception handler using Real-Time Euclid or Goodenough's notation.

Limited information passing only allows users to pass the predefined information or a limited amount of information. Ada, Windows NT, AML/X, Modula-3, Mach, and Chimera support this method. Ada provides the package *Ada.Exceptions* to save the exception name and message and to retrieve them for later analysis. Mach can only pass two fields in the exception messages. Chimera, Windows NT, Modula-3, and AML/X can only pass one argument. This argument cannot be a complex data type such as data structure. However, it can be a pointer as the case in Chimera and Windows NT. As a result, Chimera and Windows NT can still achieve unrestricted information passing except for expressive power and ease of use.

Other mechanisms have no restriction on the amount of information that can be passed. C++ and Lore define exceptions as classes. A raised exception is an instance of a class and can carry as much information as needed. Information can also be passed as parameters to an exception in the same way as passing function parameters. The cost of this method is that more memory is needed to save the information.

Information passing is a necessity for distributed exception processing, since the handler is not in the same context as the failing task and therefore cannot access the data directly. Information passing can increase the reliability of the program even in the case of nondistributed processing. Without an information-passing scheme, global variables such as *errno* in UNIX/C must be used to transmit the information. Using global variables increases module coupling and prevents the creation of reusable software components.

Methods that allow users to pass complete information without any restrictions are ideal for CB-RTS, although limited information passing is often sufficient and can better meet the performance goals. A mechanism with no information passing is not suitable for CB-RTS, since many components may be replicated, and at the very least some component-specific state must be passed to the exception handler.

## 5.6 Handler Scope

*Handler scope* is the entity to which an exception handler is attached. Depending on the exception-handling mechanisms, an entity can be a program, a procedure, a block, a statement, an expression, an object, or data. The handler scope can be divided into three categories: *local*, *global*, and *hybrid*.

The local handler scope has explicit evidence of handler affiliation with an exception activation point. For example, C++ uses the *catch* block to attach exception handlers to the *try* block. The handlers associated with the *try* block can only handle exceptions raised within the block. The *try* block is the scope of the handlers. Other examples of local scope include handlers attached to expressions, statements, and procedures. Ada, CLU, Exceptional C, Goodenough, Modula-3, and the Replacement Model all use this method.

The global handler scope does not have explicit evidence of handler association. The handler scope is a program or a process. Examples are handler association with an object, data, a process, or a program. If the handler is associated with a program, we usually consider it associated with the exception itself. Mach, AML/X, RT Euclid, Cui's mechanism, and Knudsen's mechanism use the global handler scope.

The hybrid handler scope is the combination of local and global handler scopes. In Lore, handlers can be attached to objects, the exception itself, and expressions. Chimera's mechanism can be used to achieve both local and global scopes; thus it is considered hybrid.

The disadvantage of local handler scope is that the exception-handling code is still cluttered within the main operation code, although they are separate to some degree. Another drawback is that nested blocks are usually added for the sole purpose of attaching an exception handler. As a result, it reduces the readability.

The global handler scope thoroughly solves the problem of separating exception-handling code from the main text; however, contrary to the local handler scope, it is awkward to handle the situation where different handlers are needed for the same exception at different activation points within a local scope. To deal with this, the different handlers for the same exception must have preferences or priorities in order to resolve the association in a deterministic manner.

The global handler scope also prohibits hierarchical exception handling. Hierarchical exception handling is desirable because it can achieve information hiding and encapsulation. In C++, an exception can first be handled by a low-level handler. If it cannot resolve the exception, it can rethrow the same exception, and a higher level handler can catch it. In Mach, an invoked exception reveals low-level implementation detail to the high-level handler.

Hybrid handler scope gives the user the most flexibility and combines the advantages of both global and local handler scopes. CB-RTS needs to have at least global scope to satisfy the distribution requirement, although a hybrid method is preferred; it may also provide better modular decomposition and performance by separating local and global exceptions.

## 5.7 Resource Cleanup

Programs must clean up allocated resources when they terminate, to keep the integrity, correctness, and consistency of the program. Meanwhile,

resource cleanup is also required by atomic transactions, which are important in many real-time systems. For instance, if an exception happens after one process enters a critical section, the process must leave the critical section if the failed process is going to terminate. Otherwise all processes that want to use the critical section will be blocked.

None of the existing exception-handling mechanisms have automatic resource cleanup facilities. Instead, they provide a construct which is executed whenever the guarded program unit exits, or they disable raising exceptions temporarily before releasing resources.

Windows NT, Modula-3, Lore, and Goodenough's notation use the construct method. Windows NT calls it *termination handler*. Lore implements it by using a primitive *when-exit*, and Goodenough's notation refers to it as *cleanup handler*. Programmers can release resources in this construct, and the system will execute it no matter why the unit exits, normally or exceptionally.

C++'s solution, the "resource acquisition is initialization" technique [Stroustrup 1991], is a variant of the construct method. The request of resource allocation is made only in the constructor of an object, and the allocated resources are released in the destructor. Since the destructor is called independently of whether the function exits normally or exceptionally, the resources are freed at the end of the scope of the object.

Real-Time Euclid provides the construct **initially** to prohibit exception raising from it. Although it is useful when a process is in a critical section or an initial section, this method cannot be used extensively, since it contradicts the original purpose of introducing exception handling.

Other mechanisms do not have resource cleanup support. The construct method is difficult to apply to CB-RTS without the support of the compiler. The disable method is applicable to CB-RTS, but it can result in priority inversion, as further discussed in Section 5.9. Thus it should only be used as a complementary way to preserve the consistency of the system. Investigation of alternate methods that can be incorporated into a RTOS and suitable for CB-RTS is required.

## 5.8 Exception Interface

The *exception interface* is the part in a module interface that explicitly specifies the exceptions that might be raised by the module. Exceptions undeclared in the exception interface are prohibited from propagating outside of the current module. Since exceptions introduce the possibility of nonlocal flow of control, problems such as intermodule coupling [Cui 1989] and exception safety [Custer 1993] arise. If exceptions are not listed in the headings or specifications of the module from which they are raised, their callers might not prepare for these exceptions and will create problems. An explicit exception specification solves these above problems and increases the reliability of creating an application from software components.

Another advantage of an explicit specification is that it enables static reliability checks. An explicit exception specification reduces flexibility in

favor of allowing more static checks, although a long list of potentially raised exceptions may make the whole interface of a function difficult to use. Note that reducing flexibility does not mean reducing functionality. It often means only providing one way to achieve a certain functionality, rather than 10 ways, or preventing a programmer from generating exceptions unless they are predefined, just as variables are predefined.

C++, CLU, Cui's mechanism, Goodenough's mechanism, the Replacement Model, Knudsen's mechanism, and Exceptional C specify exception interfaces in their function headings. An example of Exceptional C's function prototype is the following (see Listing 5 in the appendix for details):

```
void move (float x, float y, float orient) raises (exception err_move)
```

The part led by the key word *raises* is the exception interface of the function *move*( ). However, some mechanisms compromise the explicit specification for the sake of flexibility, especially that of the predefined exceptions that could arise anywhere. For example, in Goodenough's mechanism, a predefined exception *ENDED*, which stands for normal termination, is implicitly included in the exception interface [Goodenough 1975a].

Other mechanisms do not have any form of exception interface. CB-RTS requires the strict nature of an explicit exception interface, in order to allow for software assembly of the components [Gertz et al. 1995]. Thus the function header method may be applicable. Since a function header method requires language support, this method is not suitable for operating systems. Further research is required if an operating-system-based exception-handling mechanism is used for CB-RTS.

## 5.9 Criticality Management

*Criticality management* is the ability to dynamically change the priority or criticality of an exception handler, so that it can be changed based on the importance of the raised exception or the importance of the process in which the error occurred. If the exception is crucial to the system, its handler may execute at the highest priority, even if it is raised from a low criticality process. If it is a minor error, exception handling can be deferred.

Only the Chimera RTOS timing error detection and handling mechanism explicitly has criticality management [Stewart and Khosla 1997]. During exception handler binding, a priority is attached to the handler. When the exception handler is invoked, the priority of the process executing can be raised, lowered, or left the same. This allows critical exception handling to be executed immediately, while minor errors are handled only when higher-priority processes have completed their execution. Chimera's global error handling for nontiming errors, however, does not support criticality management.

None of the existing exception-handling mechanisms support criticality management for all types of errors. Criticality and processes are operating system concepts. However, even current operating-system-based mecha-

nisms do not support criticality management. Criticality management is an important topic for future research.

### 5.10 Reliability Checks

*Reliability checks* test for possible errors introduced by an exception-handling mechanism itself. *Static checks* are performed by the compiler while *dynamic checks* are performed by the run-time system.

Static reliability checks are useful in finding errors due to the misuse of the mechanism, such as associating more than one handler to an exception at one time. They also reduce run-time checking overhead so that the mechanism runs faster. However, if static checks are used to discover the exceptions with no handlers, they must impose some restrictions on the mechanism [Goodenough 1975a] and are burdensome to programmers [Koenig and Stroustrup 1990].

Goodenough's mechanism and the Replacement Model use static checks to ensure that no exceptions can be raised without a handler. All the other mechanisms use dynamic checks to detect the exceptions that are not handled. The program terminates when such an exception is discovered by dynamic checks.

Three approaches are used to deal with the unhandled exceptions when dynamic checks are adopted. C++, Lore, Windows NT, and Chimera use the global default handler to process any unhandled exceptions. The default handler prints out error messages and terminates the program. CLU and Exceptional C change any unhandled exceptions into a predefined exception such as **Error** or **Failure** and provide the default handler to handle the exception. The other mechanisms that use dynamic checks terminate the program directly when an unhandled exception occurs.

Both static checks and dynamic checks are desirable to maximize reliability. The use of reliability checks, however, must be balanced with the additional complexity, and the effect that complexity has on the performance goals set out in Section 4.3.

## 6. CONCLUSION

Most current research into exception-handling technologies targets non-real-time systems. There are some fundamental differences between the objectives of real-time and non-real-time systems that make it inappropriate to simply use a mechanism designed for one environment in the other environment. To summarize our findings, the following issues must be addressed in order to create an exception-handling mechanism suitable for CB-RTS.

- (1) The differences in requirements of real-time systems and non-real-time systems require a change in the design of exception handler definition and binding. Most existing mechanisms are designed such that the bulk of exception-handling overhead occurs only after an error is detected. This design is a result of one of the main goals of existing mechanisms to minimize the performance overhead of defining exceptions and

binding handlers under *normal* system operation. In real-time systems, it is acceptable to incur significant overhead during initialization, in favor of guaranteed time-bounded detection and handling. None of the mechanisms we studied, including RT Euclid and Chimera's mechanism that were designed especially for real-time systems, sufficiently address these real-time system issues.

- (2) Propagation of exceptions across process boundaries must be incorporated into an exception-handling mechanism for CB-RTS and must be transparent to the application designer. Software components in real-time systems are usually processes or threads. Interaction between components must be done through some form of interprocess communication (IPC). If an error occurs in one component and must be handled by another, then the error-handling mechanism must also use some form of IPC. Language-based mechanisms tend to not have any concept of processes, while operating system mechanisms leave IPC up to the programmer.
- (3) A mechanism that provides criticality management of hardware, software, and state errors in CB-RTS systems is needed. CB-RTS processes are typically scheduled either by a fixed or dynamic priority [Stewart 1994]. High-priority processes can preempt lower-priority processes, in order to ensure that critical processes meet their timing constraints and that critical code gets executed all the time, even in the presence of a transient overload. In the same token, exception handling of lower-priority processes must not preempt higher-priority processes from executing their normal operating code. Furthermore, if multiple exceptions are active at the same time, the highest-priority exceptions should be executed first. Only operating-system-based mechanisms had concepts of priorities, and of those, only the Chimera RTOS mechanism explicitly has criticality management for detecting and handling timing errors [Stewart and Khosla 1997].

Some requirements of CB-RTS exception handling, such as criticality management and exception raising to support distributed processing, have only been provided by operating-system-based mechanisms. On the other hand, needs such as explicit exception specification and static reliability checks have only been provided by programming-language-based mechanisms. We feel that the necessary design for an exception handler mechanism suited for CB-RTS will require a cooperative solution between the programming language and the operating system. Microsoft C plus Windows NT is currently the only such cooperative solution; however, that specific mechanism does not meet several of the essential requirements for CB-RTS.

The issues presented in this article differentiate the exception-handling needs of CB-RTS as compared to other software paradigms and can serve as a driving force for future research into exception-handling technology.

## ONLINE-ONLY APPENDIX

This appendix is available only online as a g'zipped tar file. You should be able to get the online-only appendix from the citation page for this article:

<http://www.acm.org/pubs/citations/journals/toplas/1998-20-2/p274-lang>

Alternative instructions on how to obtain online-only appendices are given on the back inside cover of current issues of TOPLAS or on the ACM TOPLAS Web page:

<http://www.acm.org/toplas>

## REFERENCES

- BARNES, J. 1995. *Programming in Ada 95*. Addison-Wesley, Reading, Mass.
- BIHARI, T. E. AND GOPINATH, P. 1992. Object-oriented real-time systems: Concepts and examples. *IEEE Comput.* 25, 12, 25–32.
- BLACK, D. L., GOLUB, D. B., HAUTH, K., TEVANIAN, A., AND SANZI, R. 1988. The Mach exception handling facility. Tech. Rep. CMU-CS-88-129, School of Computer Science, Carnegie Mellon Univ. Pittsburgh, Pa.
- BORGIDA, A. 1986. Exceptions in object-oriented languages. *ACM SIGPLAN Not.* 21, 10, 107–119.
- CAMERON, D., FAUST, P., LENKOV, D., AND MEHTA, M. 1992. A portable implementation of C++ exception handling. In *Proceedings of the USENIX C++ Technical Conference*. USENIX, Berkeley, Calif., 225–243.
- CAMPIONE, M. AND WALRATH, K. 1996. *The Java Tutorial: Object-Oriented Programming for the Internet*. The Java Series. Addison-Wesley, Reading, Mass.
- CARDELLI, L., DONAHUE, J., GLASSMAN, L., JORDAN, M., KALSOW, B., AND NELSON, G. 1988. Modula-3 report. Tech. Rep. 31, Systems Research Centre, Digital Equipment Corp., Cupertino, Calif.
- COCCO, N. AND DULLI, S. 1982. A mechanism for exception handling and its verification rules. *Comput. Lang.* 7, 2, 89–102.
- COX, I. J. AND GEHANI, N. H. 1989. Exception handling in robotics. *IEEE Comput.* 22, 3, 43–49.
- CRISTIAN, F. 1982. Exception handling and software fault tolerance. *IEEE Trans. Comput. C-31*, 6, 531–540.
- CUI, Q. 1989. Data-oriented exception handling. Ph.D. thesis, Univ. of Maryland.
- CUSTER, H. 1993. *Inside Windows NT*. Microsoft Press, Bellevue, Wash.
- DEPARTMENT OF DEFENSE. 1983. Reference manual for the Ada programming language. MIL-STD 1815A. United States Dept. of Defense, Washington, D.C.
- DONY, C. 1988. An object-oriented exception handling system for an object-oriented language. In *ECOOP '88, European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science, vol. 322. Springer-Verlag, New York.
- DONY, C. 1990. Exception handling and object-oriented programming: Towards a synthesis. *ACM SIGPLAN Not.* 25, 10, 322–330.
- ELLIS, M. AND CARROLL, M. 1995. Tradeoffs of exceptions. *C++ Rep.* 7, 3, 12 and 14, 16.
- GAUTHIER, M. 1995. Exception handling in Ada-94: Initial users' requests and final features. *ACM SIGADA Ada Lett.* 15, 1, 70–82.
- GEHANI, N. H. 1992. Exceptional C or C with exceptions. *Softw. Pract. Exper.* 22, 10, 827–848.
- GERTZ, M. W., MAXION, R. A., AND KHOSLA, P. K. 1995. Visual programming and hypermedia implementation within a distributed laboratory environment. *Int. J. Intell. Automat. Soft. Comput.* 1, 1, 43–62.
- GOLDBERG, A. AND ROBSON, D. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, Mass.

- GOODENOUGH, J. B. 1975a. Exception handling: Issues and a proposed notation. *Commun. ACM* 18, 12, 683–696.
- GOODENOUGH, J. B. 1975b. Structured exception handling. In *Conference Record of the 2nd Annual ACM Symposium on Principles of Programming Languages* 204–224.
- ISHIKAWA, Y., TOKUDA, H., AND MERCER, C. W. 1990. Object-oriented real-time language design: Constructs for timing constraints. In *Proceedings of the OOPSLA/ECOOP '90 Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM SIGPLAN Not. 25, 10, 289–298.
- KENNY, K. B. AND LIN, K.-J. 1991. Building flexible real-time systems using the Flex language. *IEEE Comput.* 24, 5, 70–78.
- KLIGERMAN, E. AND STOYENKO, A. D. 1986. Real-time Euclid: A language for reliable real-time systems. *IEEE Trans. Softw. Eng.* 12, 9, 941–949.
- KNUDSEN, J. L. 1987. Better exception-handling in block-structured systems. *IEEE Softw.* 4, 3, 40–49.
- KOENIG, A. AND STROUSTRUP, B. 1990. Exception handling for C++. In *C++ Conference Proceedings*. USENIX, Berkeley, Calif., 149–176.
- LEE, P. A. 1983. Exception handling in C programs. *Softw. Pract. Exper.* 13, 5, 389–405.
- LEVIN, R. 1977. Program structures for exceptional condition handling. Ph.D. thesis, Carnegie Mellon Univ., Pittsburgh, Pa.
- LIN, K.-J. AND NATARAJAN, S. 1988. Expressing and maintaining timing constraints in FLEX. In *Proceedings of the 9th IEEE Real-time Systems Symposium*. IEEE Computer Society Press, Los Alamitos, Calif., 96–105.
- LINDSAY, B. G. 1977. Exception processing in computer systems. Ph.D. thesis, Univ. of California, Berkeley, Calif.
- LISKOV, B. H. AND SNYDER, A. 1979. Exception handling in CLU. *IEEE Trans. Softw. Eng. SE-5*, 6, 546–558.
- MILNER, R. AND TOFTE, M. 1991. *Commentary on Standard ML*. MIT Press, Cambridge, Mass.
- MITCHELL, J. G., MAYBURY, W., AND SWEET, R. 1979. Mesa language manual. Tech. Rep. CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, Calif.
- MOON, D. A. 1974. MACLISP reference manual. MIT AI Lab., Cambridge, Mass.
- NACKMAN, L. R. AND TAYLOR, R. H. 1984. A hierarchical exception handler binding mechanism. *Softw. Pract. Exper.* 14, 10, 999–1003.
- NIEZGODA, S., HOLT, L., AND WOJCIECH, D. 1993. Some assembly required: NT's structured exception handling: The reality of structured exception handling in Windows NT may not live up to its promise. *BYTE* 18, 12, 317–322.
- NOBLE, J. M. 1968. The control of exceptional conditions in PL/I object programs. In *Proceedings of the IFIP Congress 68*. C78–C88.
- OUSTERHOUT, J. K. 1980. Partitioning and cooperation in a distributed multiprocessor operating system: Medusa. Ph.D. thesis, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pa.
- PAULSON, L. 1991. *ML for the Working Programmer*. Cambridge University Press, Cambridge, UK.
- PRADHAN, D. K. 1996. *Fault-Tolerant Computer System Design*. Prentice-Hall, Upper Saddle River, N.J.
- RASHID, R. F. AND ROBERTSON, G. G. 1981. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th ACM Symposium on Operating System Principles*. ACM, New York, 64–75.
- ROBERTS, E. S. 1989. Implementing exceptions in C. Tech. Rep. 40, Systems Research Center, Digital Equipment Corp., Cupertino, Calif.
- SCHNEIDER, S. A., CHEN, V. W., AND PARDO-CASTELLOTE, G. 1995. The ControlShell component-based real-time programming system. In *Proceedings of the IEEE Conference on Robotics and Automation*. 2381–2388.
- STANKOVIC, J. A. 1988. Misconceptions about real-time computing—A serious problem for next-generation systems. *IEEE Comput.* 21, 10, 10–19.

- STEWART, D. B. 1994. Real-time software design and analysis of reconfigurable multi-sensor based systems. Ph.D. thesis, Carnegie Mellon Univ., Pittsburgh, Pa.
- STEWART, D. B. AND KHOSLA, P. K. 1997. Mechanisms for detecting and handling timing errors. *Commun. ACM* 40, 1, 87–93.
- STEWART, D. B., SCHMITZ, D. E., AND KHOSLA, P. K. 1992a. The CHIMERA II real-time operating system for advanced sensor-based control applications. *IEEE Trans. Syst. Man Cybernet.* 22, 6, 1282–1295.
- STEWART, D. B., VOLPE, R. A., AND KHOSLA, P. K. 1992b. Integration of real-time software modules for reconfigurable sensor-based control systems. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, New York, 325–333.
- STEWART, D. B., VOLPE, R. A., AND KHOSLA, P. K. 1997. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Trans. Softw. Eng.* 23, 12.
- STRONG, D. M. AND MILLER, S. M. 1995. Exceptions and exception handling in computerized information processes. *ACM Trans. Inf. Syst.* 13, 2, 206–233.
- STROUSTRUP, B. 1991. *The C++ Programming Language*. 2nd ed. Addison-Wesley, Reading, Mass.
- TANG, L. S. 1992. A comparison of Ada and C++. In *Proceedings of the Conference for Industry, Academia and Government*. ACM Press, New York, 338–349.
- TEITELMAN, W. 1974. InterLISP reference manual. Xerox Palo Alto Research Center, Palo Alto, Calif.
- THEKKATH, C. A. AND LEVY, H. M. 1994. Hardware and software support for efficient exception handling. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*. 110–119.
- WANG, T. 1994. Better C: An object-oriented C language with automatic memory manager suitable for interactive applications. *ACM SIGPLAN Not.* 29, 12, 104–111.
- WEINREB, D. AND MOON, D. A. 1981. LISP machine manual. 4th ed. MIT AI Lab., Cambridge, Mass.
- WILENSKY, R. 1984. *Common LISPcraft*. Norton.
- WOLFE, V., DAVIDSON, S., AND LEE, I. 1991. RTC: Language support for real-time concurrency. In *Proceedings of the Real-Time Systems Symposium—1991*. IEEE Computer Society Press, Los Alamitos, Calif., 43–52.
- YEMINI, S. AND BERRY, D. M. 1985. A modular verifiable exception-handling mechanism. *ACM Trans. Program. Lang. Syst.* 7, 2, 214–243.

Received December 1996; revised September 1997; accepted November 1997