Advanced Exception Handling Mechanisms

Peter A. Buhr and W. Y. Russell Mok

Abstract—It is no longer possible to consider exception handling as a secondary issue in language design, or even worse, a mechanism added after the fact via a library approach. Exception handling is a primary feature in language design and must be integrated with other major features, including advanced control flow, objects, coroutines, concurrency, real-time and polymorphism. Integration is crucial as there are both obvious and subtle interactions between exception handling and other language features. Unfortunately, many exception handling mechanisms work only with a subset of the features and in the sequential domain. A framework for a comprehensive, easy to use, and extensible exception handling mechanism is presented for a concurrent, object-oriented environment. The environment includes language constructs with separate execution stacks, e.g., coroutines and tasks, so the exception environment is significantly more complex than the normal single-stack situation. The pros and cons of various exception features are examined, along with feature interaction with other language mechanisms. Both exception termination and resumption models are examined in this environment, and previous criticisms of the resumption model, a feature commonly missing in modern languages, are addressed.

Index Terms—Exception handling, robustness, termination, resumption, concurrent, interrupts, object-oriented.

1 Introduction

SUBSTANTIAL research has been done on exceptions but there is hardly any agreement on what an exception is. Attempts have been made to define exceptions in terms of errors but an error itself is also ill-defined. Instead of struggling to define what an exception is, this paper examines the entire process as a control flow mechanism, and an exception is a component of an exception handling mechanism (EHM), which specifies program behaviour after an exception has been detected. The control flow generated by an EHM is supposed to make certain programming tasks easier, in particular, writing robust programs.

Prior to EHMs, the common programming techniques used to handle exceptions were return codes and status flags. The *return code* technique requires each routine to return a value on its completion. Different values indicate if a normal or rare condition has occurred during the execution of a routine. Alternatively, or in conjunction with return codes, is the *status flag* technique, which uses a shared variable to indicate the occurrence of a rare condition. Setting a status flag indicates a rare condition has occurred; the value remains as long as it is not overwritten.

The authors are with the Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1.
E-mail: {pabuhr,wyrmok}@uwaterloo.ca.

Manuscript received 28 Feb. 1999; revised 05 Dec. 1999; accepted 28 Jan. 2000. Recommended for acceptance by A. Romanovshy.

Appears in IEEE TRANSACTION ON SOFTWARE ENGINEERING, VOL. 26, NO. 9, SEPTEMBER 2000

Both techniques have noticeable drawbacks. First, and foremost, the programmer is required to explicitly test the return values or status flags; hence, an error is discovered and subsequently handled only when checks are made. Without timely checking, a program is allowed to continue after an error, which can lead to wasted work, at the very least, and an erroneous computation at worst. Second, these tests are located throughout the program, reducing readability and maintainability. Third, as a routine can encounter many different errors, it may be difficult to determine if all the necessary error cases are handled. Finally, removing, changing or adding return or status values is difficult as the testing is coded inline. The return code technique often encodes exception values among normal returned values, which artificially enlarges the range of valid values independent of the computation. Hence, changing a value representing an exception into a normal return value or vice versa may be difficult as such changes affect existing programs. The status flag technique uses a shared variable that precludes its use in a concurrent environment as it can change unpredictably.

This paper discusses modern EHM techniques, which are finally supplanting return codes and flags even though EHMs have been available for more than two decades. A general framework is presented for exception handling, along with an attempt to compose an ideal EHM, with suggested solutions to some outstanding EHM problems. In constructing the framework, a partial survey of existing EHMs is necessary to compare and contrast approaches.

2 EHM Objectives

The failure of return codes and status flags indicates the need for an EHM, which must:

1. alleviate testing for the occurrence of rare conditions throughout the program, and from explicitly changing the control flow of the program,
2. provide a mechanism to prevent an incomplete operation from continuing,
3. be extensible to allow adding, changing and removing exceptions.

The first objective targets readability and programmability by eliminating checking of return codes and flags. The second objective provides a transfer from the exception point that disallows returning, directing control flow away from an operation where local information is corrupted, i.e., the operation is *non-resumable*. The last objective targets extensibility, easily allowing change in the EHM, and these changes should have minimal effects on existing programs using them.

Two EHM examples illustrate the three objectives:

- **Unix signal mechanism.** On encountering a rare condition, a signal (interrupt) is generated, which preempts execution and calls a handler routine, suspending prior execution; when the handler routine returns, prior execution continues. This change of control flow does not require the programmer's involvement or testing any error code, as there is no explicit call in the program. Using a non-local jump facility, longjmp, the handler routine can prevent an incomplete operation from continuing, and possibly terminate multiple active blocks between the signal handler and the non-local transfer point (see also Section 5.1). Extensibility is quite limited, as most signals are predefined and unavailable to programmers. If a library uses one of the few user available signals, all clients must agree on the signal's definition, which may be impossible.

- **Ada exception mechanism.** On encountering a rare condition, an exception is *raised* in Ada terminology, and control flow transfers to a sequence of statements to handle the exception. This change of control flow does not require the programmer's involvement or testing any error code. The operation encountering the rare condition cannot be continued, and possibly multiple active blocks between the raise point and the statements handling the exception are terminated. New exceptions can be declared as long as there is no name conflict in the flat exception name-space; hence the mechanism is extensible.

3 Execution Environment

The structure of the execution environment has a significant effect on an EHM, e.g., concurrency requires a more complex EHM than a sequential environment. The execution model described in [1] is adopted for this discussion; it identifies three elementary execution properties:

- **Execution.** is the state information needed to permit independent execution. It includes local and global data, current execution location, and routine activation records of a sequential computation.
- **Thread.** is execution of code that occurs independently of and possibly concurrently with another execution; thread execution is sequential as it changes an execution's state. Multiple threads provide *concurrent execution*; multiple CPUs provide *parallel execution* of threads. A *context switch* is a change in the execution/thread binding.
- **Mutual exclusion.** is serializing execution of an operation on a shared resource.

The first two properties are fundamental, i.e., it is impossible to create them from simpler constructs in a programming language. Only mutual exclusion can be generated using basic control structures and variables (e.g., Dekker's algorithm), but software algorithms are complex and inefficient. Thus, these 3 properties must be supplied via the programming language. Different combinations of the execution properties generate all of the higher-level language constructs present in existing languages, e.g., routine, class, coroutine, monitor, and task. Both coroutine and task have separate executions; only a task has a thread. An EHM affects the control flow of an execution, and therefore, it is often sufficient when discussing exceptions to refer to an execution without knowing if it is a coroutine or a task. A monitor, a coroutine, and a task can have mutual exclusion.

4 EHM Overview

An *event* is an exception instance, and is raised by executing a language or system operation, which need not be available to programmers, e.g., only the runtime system may raise predefined exceptions. Raising an exception indicates an abnormal condition the programmer cannot or does not want to handle via conventional control flow. What conditions are abnormal is programmer determined. The execution raising the event is the *source execution*.

The execution that changes control flow due to a raised event is the *faulting execution*; its control flow is routed to a handler. With multiple executions, it is possible to have an exception raised in a source execution different from the faulting execution. *Propagating* an exception directs the control flow of the faulting execution to a handler, and requires a *propagation mechanism* to locate the handler. The chosen handler is said to have *caught* the event when execution transfers there. A *handler* is a sequence of statements dealing with a set of exceptions. The faulting execution *handles* an event by executing a handler associated with the raised exception. It is possible that another exception is raised while executing the handler. A handler is said to have *handled* an event only if the handler returns. Unlike returning from a routine, there may be multiple return mechanisms for a handler (see Section 5).

For a *synchronous* exception, the source and faulting execution are the same, i.e., the exception is raised and handled by the same execution. It is usually difficult to distinguish raising and propagating in the synchronous case, as both happen together. For an *asynchronous* exception, the source and faulting execution are usually different, e.g., raise E in Ex raises exception E from the current source to the faulting execution Ex. Unlike a synchronous exception, raising an asynchronous exception does not lead to the immediate propagation of the event in the faulting execution. In the Unix example, an asynchronous signal can be blocked, delaying propagation in the faulting execution. Rather, an asynchronous exception is more like a non-blocking direct communication from the source to the faulting execution. The change in control flow in the faulting execution is the result of *delivering* the exception event, which initiates the propagation of the event in the faulting execution. While the propagation in the faulting execution can be carried out by the source, faulting or even another execution, for the moment, assume the source raises the event and the faulting execution propagates and handles it.

Goodenough's seminal paper on exception handling suggests a handler can be associated with programming units as small as a sub-expression and as large as a routine [2, pp. 686-687]. Between these extremes is associating a handler with a language's notion of a block, i.e., the facility that combines multiple statements into a single unit, as in Ada [3], Modula-3 [4] and C++ [5]. While the granularity of a block is coarser than an expression, our experience is that fine grained handling is rare. As well, having handlers, which may contain arbitrarily complex code, in the middle of an expression can be difficult to read. Finally, handlers in expressions or for routines may need a mechanism to return results to allow execution to continue, which requires additional constructs [2, p. 690].

In addition, a handler can handle multiple exceptions and multiple handlers can be bound to one block. Syntactically, the set of handlers bound to a particular block is the *handler clause*, and a block with handlers becomes a *guarded block*. A block with no handler clause is an *unguarded block*. An exception can propagate from any block; the propagation mechanism determines the order the handler clauses bound to different blocks are searched.

5 HANDLING MODELS

Yemini and Berry [6, p. 218] identify 5 exception handling models: non-local transferring, 2 terminating, retrying and resuming. An EHM can provide multiple models.

5.1 Non-local Transfer

PL/I [7] is one of a small number of languages that supports non-local transfer among dynamic blocks through the use of label variables. A label variable in PL/I contains both a point of transfer and a pointer to an activation record on the stack containing the transfer point; therefore, a label variable is not a static object. The non-local transfer in PL/I directs control flow to the label variable specified in a goto statement. A consequence of the transfer is that blocks activated after the one with the label variable are destroyed. The C routines setjmp and longjmp are a simplified non-local transfer, where setjmp sets up a dynamic label variable and longjmp performs the non-local transfer.

An EHM can be constructed using a non-local transfer mechanism by labelling code to form handlers and terminating operations with non-local transfers to labels in prior blocks. However, non-local transfer is too general, allowing branching to almost anywhere (the structured programming problem). This lack of discipline makes programs less maintainable and error-prone [8, p. 102]. More importantly, an EHM is essential for sound and efficient code generation by a compiler (as for concurrency [9]). If a compiler is unaware of exception handling (e.g., setjmp/longjmp in C), it may perform code optimizations that *invalidate* the program, needing bizarre concepts like the volatile declaration qualifier. Hence, this model is rejected.

5.2 Termination Model

In the termination model, control flow transfers from the raise point to a handler, terminating intervening blocks; when the handler completes, control flow continues as if the incomplete operation in the guarded block terminated without encountering the exception. Hence, the handler acts as an alternative operation for its guarded block. This model is the most popular, appearing in Ada, C++, ML [10], Modula-3 and Java [11].

5.3 Retrying Model

The retry model combines the termination model with special handler semantics, i.e., restart the failed operation. There must be a clear beginning for the operation to be restarted. The beginning of the guarded block is usually the restart point and there is hardly any other sensible choice. Figure 1 gives an example by extending C++ with a retry handler. The exception Negative is raised using termination semantics, and the retry handler completes by jumping to the start of the try block. The handler is supposed to remove the abnormal condition so that the operation can complete during retry. Mesa [12], Exceptional C [13] and Eiffel [14] provide retry semantics through a retry statement only available in the handler body.

```
1 void f( int a[ ] ) {
2     try {
3         int sum = 0;
4         for ( int i = 0; sum > 100; i += 1 ) {
5             if ( a[i] < 0 ) throw Negative(i);
6             sum += a[i];
7         }
8     } retry( Negative(p) ) a[p] = 0;
9 }
```

Fig. 1. Retry Handling

As mentioned, establishing the operation restart point is essential; reversing the second and third lines of f in the figure generates a subtle error with respect to the exception but not normal execution, i.e., the sum counter is not reset on retry. This error can be difficult to discover because control flow involving propagation occurs infrequently. In addition, when multiple handlers exist in the handler clause, these handlers must use the same restart point, which may make retrying more difficult to use in some cases. Finally, Gehani [13, p. 834] shows the retrying model can be mimicked with a loop and the termination model. We believe mimicking is superior so all looping is the result of language looping constructs, not hidden in the EHM. Because of the above problems and that retry can be mimicked easily with termination and looping, this model is rejected.

5.4 Resumption Model

In the resuming model, control flow transfers from the raise point to a handler to correct the incomplete operation, and then back to the raise point. Indeed, control flow into and out of the handler is identical to that of a routine call [13], [15]. The difference between a normal routine call and a resuming call is that the resuming call locates the handler dynamically. While resuming handlers can be simulated by passing routines as arguments that are subsequently called at the raise point, the simulation is impossible for legacy code and prevents reuse (see Section 6.5).

Liskov/Snyder [16, p. 549], and Mitchell/Stroustrup [17, p. 392] argue against the resumption model but the reasons seem anecdotal. Goodenough's resumption model is complex and Mesa's resumption is based on this model [6, pp. 235-240]. However, a resumption model can be as simple as dynamic routine call, which is easy to implement in languages with nested routines. For languages without nested routines, like C/C++, it is still possible to construct a simple resumption model [13], [18], [19]. Given a simple resumption model, the only remaining problem is recursive resumption, which is discussed in Section 8.1.3.

6 EHM FEATURES

This section examines additional features that make an EHM easy to use (see also [2], [15], [18], [20]).

6.1 Catch-Any and Reraise

It is important to have a simple mechanism for catching any exception entering a guarded block, and to subsequently reraise the unknown exception, e.g.:

```
try { ...
} catch(...) {              // catch any exception
    ... raise;              // reraise unknown exception
}
```

For termination, this capability still allows clean up of a guarded block when it does not handle an exception. Block finalization executed on *both* normal and exceptional termination, and C++-style object destructors also allow clean up for this case. For resumption, this capability allows a guarded block to gather or generate information about control flow passing through the guarded block.

6.2 Derived Exceptions

An exception hierarchy is useful to organize exceptions, similar to a class hierarchy in object-oriented languages. An exception can be derived from another exception, just like deriving a subclass from a class. A programmer can then choose to handle an exception at different degrees of specificity along the hierarchy; hence, derived exceptions support a more flexible programming style.

An important design question is whether to allow derivation from multiple exceptions, called *multiple derivation*, which is similar to multiple inheritance of classes. While Cargill [21] and others argue against multiple inheritance as a general programming facility, the focus here is on derived exceptions. Consider the following example of multiply deriving an exception [20]:

```
exception network-err, file-err;
exception network-file-err : network-err, file-err; // multiple derivation
```

While this looks reasonable, there are subtle problems:

```
try { ... raise network-file-err ...
} catch( network-err )      // close network connection
  catch( file-err )         // close file
```

If network-file-err is raised, neither of the handlers may be appropriate to handle the raised event, but more importantly, which handler in the handler clause should be chosen because of the inheritance relationship? Executing both handlers may look legitimate, but indeed it is not. If a handler clause has a handler only for file-err, does it mean that it cannot handle network-file-err completely and should raise network-err afterwards? The example shows that handling an exception having multiple parents may be inappropriate. If an exception cannot be caught by one of its parent, the derivation becomes moot. Therefore, multiple derivation is rejected for derived exceptions as it introduces significant complications into the semantics.

6.3 Exception Parameters

The ability to pass data from the source to the faulting execution is essential for the handler to analyse why an exception was raised and how to deal with it. *Exception parameters* enable the source to transfer important information into and out of a handler, just like routine parameters and results. An exception parameter can be read-only, write-only and read-write. While information could be passed through shared objects, exception parameters eliminate side-effects and locking in a concurrent environment. Ada has no parameters, Modula-3 has a single parameter, C++ has an object (essentially a single parameter), ML and Mesa have multiple parameters.

Parameter specification for an exception depends on the form of the exception declaration. In Mesa and Modula-3, a technique similar to routine parameters is used, as in:

```
exception E( int );     // exception declaration with parameter
raise E( 7 );           // argument supplied at raise
catch( E( p ) )         // handler receives argument in parameter
```

In C++, an object type is the exception and an object instance is created from it as the parameter, as in:

```
struct E { int p; E(int p) : p(p) {} };
throw E( 7 );           // create object, initialized with 7
catch( E p ) ...        // handler receives object in parameter
```

In all cases, it is possible to have parameters that are routines (or member routines), and these routines can perform special operations. For example, by convention or with special syntax, an argument or member routine can be used as a *default handler*, which is called if the faulting execution does not find a handler during propagation, as in:

```
exception E( ... ) default( f ); // default routine f
struct E { ... void default() {...}; }; // named default member
```

Other specialized operations are conceivable.

The arguments of an asynchronous exception are usually accessible to the source after the event is raised, and to the faulting execution after the event is caught. Therefore, access to these arguments must be properly synchronized in a concurrent environment if pointers are involved. The synchronization can be provided by the EHM or by the programmer. The former makes programming easier but can lead to unnecessary synchronization as it requires blocking the source or the faulting execution when the argument is accessed, which may be inappropriate in certain cases. The latter is more flexible as it can accommodate specific synchronization needs. With the use of monitors, futures, conditional variables and other facilities for synchronization, the synchronization required for accessing an exception argument can be easily implemented by a programmer. Hence, leaving synchronization to the programmer simplifies the EHM interface and hardly loses any capabilities.

Finally, with derived exceptions, parameters to and results from a handler must be dealt with carefully depending on the particular language. For example, given exception D, derived from B, with additional data fields for passing information into and out of a handler, when a D event is raised and caught by a handler bound to B, it is being treated as a B exception within the handler and the additional data fields cannot be accessed without a dynamic down-cast. Consequently, if the handler returns to the raise point, some data fields in D may be uninitialized. A similar problem occurs if static dispatch is used instead of dynamic (both Modula-3 and C++ support both forms of dispatch). The handler treating exception D as a B can call members in B with static dispatch rather than members in D. For termination, these problems do not exist because the handler parameters are the same or up-casts of arguments. For resumption, any result values returned from the handler to the raise point are the same or down-casts of arguments. The problem of down-casting is a subtyping issue, independent of the EHM, which programmers must be aware of when combining derived exceptions and event parameters with resumption.

6.4 Bound Exceptions and Conditional Handling

In Ada, an exception declared in a generic package creates a new instance for each package instantiation, e.g.:

```
    generic package Stack is
        overflow : exception; ...
    end Stack;
    package S1 is new Stack;      -- new overflow
    package S2 is new Stack;      -- new overflow
begin
    ... S1.push(...); ... S2.push(...); ...
exception
    when S1.overflow => ...       -- catch overflow for S1
    when S2.overflow => ...       -- catch overflow for S2
```

Hence, it is possible to distinguish which stack raised the overflow without passing data from the raise to the exception. In object-oriented languages, the class is used as a unit of modularity for controlling scope and visibility. Similarly, it makes sense to associate exceptions with the class that raises them, as in:

```
class file {
    exception file-err; ...
```

However, is the exception associated with the class or objects instantiated from it? As above, the answer affects the capabilities for catching the exception, as in:

```
file f;
try { ... f.read(...); ...          // may raise file-err
} catch( file::file-err ) ...       // option 1
  catch( f.file-err ) ...           // option 2
```

In option 1, only one file-err exception exists for *all* objects created by type file. Hence, this catch clause deals with file-err events regardless of which file object raises it. In option 2, each file object has its own file-err exception. Hence, this catch clause *only* deals with file-err events raised by object f, i.e., the handler is for an event bound to a particular object, called a *bound exception*. This specificity prevents the handler from catching the same exception bound to a different object. Both facilities are useful but the difference between them is substantial and leads to an important robustness issue. Finally, an exception *among* classes is simply handled by declaring the exception outside of the classes and referencing it within the classes.

Bound events cannot be trivially mimicked by other mechanisms. Deriving a new exception for each file object (e.g., f-file-err from file-err) results in an explosion in the total number of exceptions, and cannot handle dynamically allocated objects, which have no static name. Passing the associated object as an argument to the handler and checking if the argument is the bound object, as in:

```
catch( file::file-err( file *fp ) ) // fp is passed from the raise
    if ( fp == &f ) ...      // deal only with f
    else raise               // reraise event
```

requires programmers to follow the coding convention of reraising the event if the bound object is inappropriate [18]. Such a coding convention is unreliable, significantly reducing robustness. In addition, mimicking becomes infeasible for derived exceptions, as in:

```
exception B( obj );          // base exception
exception D( obj ) : B;      // derived exception
obj o1, o2;
try { ... raise D(...); ...
} catch( D( obj *o ) )       // deal with derived exception
    if ( o == &o1 ) ...      // deal only with o1
    else raise               // reraise event
  catch( B( obj *o ) )       // deal with base exception
    if ( o == &o2 ) ...      // deal only with o2
    else raise               // reraise event
```

Assuming exception D is raised, the problem occurs when the first handler catches the derived exception and reraises it if the object is inappropriate, because the reraise now precludes the handler for the base exception from being chosen as a handler clause has been selected for the guarded block. Therefore, the "catch first, then reraise" approach is an incomplete substitute for bound exceptions.

Finally, it is possible to generalize the concept of the bound exception with *conditional handling* [22], as in:

```
catch( E( obj &o ) ) when( o.f == 5 ) ...
```

where the when clause specifies a general conditional expression that must also be true before the handler is chosen. Conditional handling can mimic bound events simply by checking if the object parameter is equal to the desired object. Also, the object in the conditional does not have to be the object containing the exception declaration as for bound exceptions. The problem with conditional handling is the necessity of passing the object as an argument or embedding it in the exception before it is raised. Furthermore, there is now only a coincidental connection between the exception and conditional object versus the statically nested exception in the bound object. While we have experience on the usefulness of bound exceptions [18], we have none on conditional handling.

6.5 Exception List

An *exception list* is part of a routine's signature and specifies which exceptions may propagate to its caller, e.g., in Goodenough, CLU [16], Modula-3 and Java (optional in C++). In essence, the exception list is precisely specifying the behaviour of a routine. The exception list allows an exception to propagate through many levels of routine call only as long as it is explicitly stated that propagation is allowed. This capability allows static detection of situations where a raised exception is not handled locally or by its caller, or runtime detection where the exception may be converted into a special failure exception or the program terminated.

While specification of routine behaviour is certainly essential, this feature is too restrictive [17, p. 394], having a significant feature interaction between the EHM and a language's type system. For example, consider the C++ template routine sort:

```
template<class T> void sort( T items[ ] ) {
    // using bool operator<( const T &a, const T &b );
```

using the operator routine <. In general, it is impossible to know which exceptions may be propagated from the routine <, and subsequently those from sort because sort takes many different < routines to support code reuse. Therefore, it is impossible to give an exception list on the template routine. An alternative is to add the specification at instantiation of the template routine, as in:

```
sort( V ) raises X, Y, Z; // instantiate with exception list
```

This case works because a new sort routine is essentially generated for each instantiation and so there are actually multiple versions each with a different signature. However, if sort is precompiled or passed as a further argument, there is only one signature to match all calls.

As well, for arguments of routine pointers (functional style) and/or polymorphic methods or routines (object-oriented style), exception lists preclude reuse, e.g.:

```
int f( int (*g)(...) ) {
    ... *g(...) ...
}
int g() raises(E) { raise E; }
int h(...) {
    try { ... f( g ); ...
    } catch( E ) ...
}
```

```
class B {
    virtual int g() {}
    int f() { ... g(); ... }
};
class D : public B {
    int g() raises(E) { raise E; }
    int h() {
        try { ... f(); ...
        } catch( E ) ...
    }
};
```

The left example illustrates arguments of routine pointers, where routine h can handle exception E raised by g, but only if it passed unchanged through the intermediate routine f. Similarly, the right example illustrates object-oriented dynamic dispatch, where the derived class replaces member g, which is called from member B::f. Member routine D::h calls B::f, which calls D::g with the potential to raise exception E. Member D::h is clearly capable of handling the exception because it created the version of g raising the event. However, this reasonable case is precluded because the signature of D::g is less restrictive than B::g. If f in the left example or B in the right example are precompiled in a library, there is no option to expand the signatures to allow this reuse scenario. Nor is it reasonable to expand the signature for every routine. In fact, doing so makes the program less robust because the signature now covers too broad a range of exceptions. Converting the specific raised exception to the failure exception at the boundary where the specific exception does not appear in the exception list precludes any chance of handling the specific event at a lower level and only complicates any recovery. The problem is exacerbated when a raised event has an argument because the argument is lost in the conversion. Finally, determining an exception list for a routine becomes difficult or impossible with the introduction of asynchronous exceptions because an asynchronous exception may be propagated at any time.

7 Propagation Mechanisms

Propagating directs control flow of the faulting execution to a handler; the search for a handler normally proceeds through the blocks, guarded and unguarded, on the runtime stack. Different implementation actions occur during the search depending on the kind of propagation. Two kinds of propagation are possible, throwing and resuming, corresponding to the termination and resumption models, respectively, and both forms can coexist in a single EHM.

Throwing propagation means control does not return to the point of the raise. This semantics implies the blocks on the stack between the raise and the handler/guarded block are destroyed, called *stack unwinding*. The unwinding associated with throwing normally occurs during the propagation, although this is not required; unwinding can occur when the handler is found, during the handler's execution, or on its completion. However, there is no advantage to delaying unwinding for throwing, and doing so results in problems (see Section 9.2) and complicates most implementations.

Resuming propagation means control returns to the point of the raise; hence, there is no stack unwinding. However, a handler may determine that control cannot return, and need to unwind the stack, i.e., change the resume into a throw. One mechanism to allow this capability is an unwind statement available in the handler to trigger stack unwinding, as in VMS [23]. With an unwind statement, it is conceivable to eliminate throwing and have only resuming propagation, where the handler determines whether to unwind the stack. Unfortunately, the absence of throwing precludes the source from forcing unwinding, and consequently, opens the door for unsafe resumption.

In an EHM where throwing and resuming coexist, it is possible to partially override their semantics by raising events within a handler, as in:

```
try { ... resume E1;
} catch( E1 ) throw E2;
```

```
try { ... throw E1;
} catch( E1 ) resume E2;
```

In the left example, the throw overrides the resuming and forces stack unwinding, starting with the stack frame of the handler (frame on the top of the stack), followed by the stack frame of the block that originally resumed the exception. In the right example, the resume cannot override the throwing because the stack frames are already unwound, so the new resume starts with the handler stack frame.

Two approaches for binding throwing or resuming propagation to an exception are available: binding when declaring the exception or when raising the event. Associating the propagation mechanism at exception declaration forces a partitioning of exceptions, as in Goodenough [2] with ESCAPE and NOTIFY, μSystem [18] with exceptions and interventions, and Exceptional C [13] with exceptions and signals. A single overloaded raise statement is sufficient as the exception determines the propagation mechanism. Alternatively, the exception declaration is general and binding occurs when raising the event, which requires a throw and a resume statement to indicate the particular mechanism. Hence, the same exception can be thrown or resumed at different times. We know of no actual EHM allowing general exceptions to be either thrown or resumed.

8 Propagation Models

The propagation mechanism determines how to find a handler, and most EHMs adopt *dynamic propagation*, which searches the call stack to find a handler. The other propagation mechanism is *static propagation*, which searches the lexical hierarchy. Static propagation was proposed by Knudsen [24], [25], and his work has been largely ignored in the EHM literature. As a result, dynamic propagation is often known as propagation.

8.1 Dynamic Propagation

Dynamic propagation allows the handler clause bound to the top block on the call stack to handle the event, provided it has an appropriate handler. A consequence is that the event is handled by a handler closest to the block where propagation of the event starts. Usually, operations higher on the stack are more specific while those lower on the call stack are more general. Handling an exception at the highest level deals with the exception in a context that is more specific, without affecting the abstract operation at a lower level. Handling an exception is often easier in a specific context than in a general context. Dynamic propagation also minimizes the amount of stack unwinding when (dynamically) throwing an event.

However, there are criticisms against dynamic propagation: visibility problem, dynamic handler selection, and recursive resuming. These criticisms are discussed before looking at static propagation, a proposal intended to solve the problems of dynamic propagation.

8.1.1 Visibility

Dynamic propagation can propagate an exception into a block in a different lexical scope, as in the examples in Section 6.5. In this case, the exception is propagated through a scope where it is invisible and then back into a scope where it is visible. It has been suggested this semantics is undesirable because a routine is indirectly propagating an exception it does not know [26]. Some language designers believe an exception should never be is propagated into a scope where it is invisible, or if allowed, the exception should lose its identity and be converted into a general failure exception. However, we have demonstrated the reuse restrictions resulting from complete prevention and loss of specific information for conversion when these semantics are adopted (see end of Section 6.5).

8.1.2 Dynamic Handler Selection

With dynamic propagation, the handler chosen for an exception cannot usually be determined statically, due to conditional code or calls to precompiled routines raising an event. Hence, a programmer seldom knows statically which handler is selected, making the program more difficult to trace and the EHM harder to use [6], [8], [24], [26].

However, when raising an exception it is rare to know what specific action is taken; otherwise, it is unnecessary to define the handler in a separate place, i.e., bound to a guarded block lower on the call stack. Therefore, the uncertainty of a handling action when an event is raised is not introduced by a specific EHM but by the nature of the problem and its solution. For example, a library normally declares exceptions and raises them without providing any handlers; the library client provides the specific handlers for the exception in their applications. Similarly, the return code technique does not allow the library writer to know the action taken by a client. When an EHM facility is used correctly, the control flow of propagation and the side-effects of handlers should be understandable.

8.1.3 Recursive Resuming

Because resuming propagation does not unwind the stack, handlers defined in previous scopes continue to be present during resuming propagation. In throwing propagation, the handlers in previous scopes disappear as the stack is unwound. The presence of resuming handlers in previous scopes can cause a situation called *recursive resuming*. The simplest situation where recursive resuming can occur is when a handler for a resuming exception resumes the same event, as in:

```
try { ... resume R; ...      // T(H(R)) => try block handles R
} catch( R ) resume R;       // H(R) => handler for R
```

The try block resumes R. Handler H is called by the resume, and the blocks on the call stack are:

```
... → T(H(R)) → H(R)
```

Then H resumes exception R again, which finds the handler just above it at T(H(R)) and calls handler H(R) again and this continues until the runtime stack overflows. Recursive resuming is similar to infinite recursion, and is difficult to discover both at compile time and at runtime because of the dynamic choice of a handler. Asynchronous resuming compounds the difficulty because it can cause recursive resuming where it is impossible for synchronous resuming.

MacLaren briefly discussed the recursive resuming problem in the context of PL/I [8, p. 101], and the problem exists in Exceptional C and μSystem. Mesa made an unsuccessful attempt to solve this problem because its solution is often criticized as incomprehensible. The Mesa and other possible solutions are discussed in Section 14.

8.2 Static Propagation

Knudsen proposed a *static propagation* mechanism [24], [25], with the intention of resolving the dynamic propagation problems, using a handler based on Tennent's *sequel* construct [27, p. 108]. A *sequel* is a routine, including parameters; however, when a sequel terminates, execution continues at the end of the block in which the sequel is declared rather than after the sequel call. Thus, handling an exception with a sequel adheres to the termination model. However, propagation is along the lexical hierarchy, i.e., static propagation, because of static name binding. Hence, for each sequel call, the handling action is known at compile time. Finally, Knudsen augments the sequel with virtual and default sequels to deal with controlled cleanup, but points out that mechanisms suggested in Section 6.1 can also be used [25, p. 48].

While static propagation is feasible for monolithic programs, it fails for modular (library) code as the static context of the module and user code are disjoint, e.g.:

```
{ // new block
    sequel StackOverflow(...) { ... }
    class stack {
        void push( int i ) { ... raise StackOverflow(...); }
        ...
    };
    stack s;
    ... s.push( 3 ); // causes overflow
} // sequel transfers to end of lexical scope
```

if stack is separately compiled, StackOverflow cannot be referenced in the user's code. To overcome this problem, a sequel can be made a parameter of stack, e.g.:

```
class stack {       // separately compiled
    stack( sequel overflow(...) ) { ... } // constructor
    void push( int i ) { ... raise overflow(...); }
};
{                   // separately compiled
    sequel StackOverflow(...) { ... }
    stack s( StackOverflow );
    ... s.push( 3 ); // causes overflow
} // sequel transfers to end of lexical scope
```

In static propagation, every exception raised during a routine's execution is known statically, i.e., the static context and/or sequel parameters form the equivalent of an exception list (see Section 6.5). However, when sequels become part of a class's or routine's type signature, reuse is inhibited, as for exception lists. Furthermore, declarations and calls now have potentially many additional arguments, even if parameter defaults are used, which results in additional execution cost on every call. Interestingly, the dynamic handler selection issue is resolved only for monolithic programs; when sequels are passed as arguments, the selection becomes dynamic, i.e., the raise point does not know statically which handler is chosen, but it does eliminate the propagation search. Finally, there is no recursive resuming because there is no special resumption capability; resumption is achieved by explicitly passing fix-up routines and using normal routine call, which is available in most languages. However, passing fix-up routines has the same problems as passing sequel routines. Essentially, if users are willing to explicitly pass sequel arguments, they are probably willing to pass fix-up routines.

Finally, Knudsen shows several examples where static propagation provides syntax and semantics superior to traditional dynamic EHMs (e.g., CLU/Ada). However, with advanced language features like generics and overloading, and advanced EHM features suggested in this paper, it is possible to achieve almost equivalent syntax and semantics in all cases. For these reasons, static propagation is rejected for an EHM, in favour of the more powerful and expressive dynamic propagation.

9 HANDLER CONTEXT

The static context of a handler is examined with respect to its guarded block and lexical context.

9.1 Guarded Block

The static context of handlers is different in C++ and Ada. A C++ handler executes in a scope outside its guarded block, while an Ada handler is nested inside its guarded block, and hence, can access variables declared in it, e.g.:

```
int x; // outer                VAR x : INTEGER; -- outer
try {                          BEGIN
    int x; // inner                VAR x : INTEGER; -- inner
} catch( ... ) {               EXCEPTION WHEN Others =>
    x = 0; // outer x              x := 0; -- inner x
}                              END;
```

By moving the handler and possibly adding another nested block, the same semantics can be accomplished in either language, as long as a handler can be associated with any nested block. According to [20], the approach in C++ can lead to better use of registers. Because one approach can mimic the other, local declarations in a guarded block are assumed to be invisible in the handler.

9.2 Lexical Context

Resuming a handler is like calling a nested routine, which requires the lexical context for the handler to access local variables in its static scope. In general, languages with nested routines (or classes) use lexical links among activation records, which are traversed dynamically for global references. Compilers often attempt to optimize lexical links for performance reasons, which can complicate resumption. For termination, when the stack is unwound immediately, lexical links are unnecessary. If unwinding occurs during or after the handler is executed, it may be necessary to ignore extraneous stack contexts, which would cause incorrect references or recursive resuming.

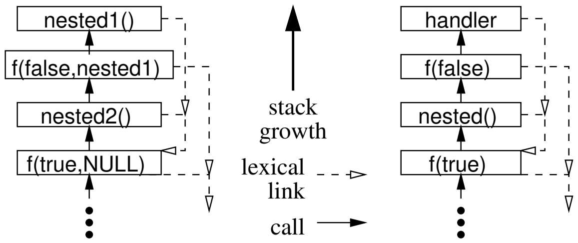


Fig. 2. Lexical Contexts

The lexical context for resuming handlers has been cited as a source of confusion and complexity [13, p. 833] [17, pp. 391-392], which occurs with either statically or dynamically bound handlers. Confusion results from unexpected values being accessed due to differences between static and dynamic contexts, and complexity from the need for lexical links. However, both these issues are related to nesting and not specific to an EHM. The following examples generate identical dynamic situations (see Figure 2):

```
void f( bool x, void (*r)() ) {        void f( bool x ) {
    void nested1() {...}                    void nested() {
    void nested2() {                            try { f( ! x );
        f( ! x, nested1 );                      } catch( E ) ...;
    }                                       }
    if ( x ) nested2();                     if ( x ) nested();
    else r();                               else resume E;
}                                       }
```

The call to nested1 in the left example and the resuming handler in the right example both have a lexical context of f(true...), so both routines reference x with a value of true even though there is an instance of f(false...) (i.e., x is false) directly above them on the stack. As mentioned, this confusion and lexical links are an artifact of nesting not the resumption model. We believe the number of cases where this problem occurs are few. In languages without nested routines, e.g., C/C++, these issues do not exist, but the resuming handlers must then be specified separately from the guarded block, affecting readability [13], [18], [19].

10 HANDLER PARTITIONING

An EHM can force a handler to bind to one handling model at compile time by declaring a *terminating handler* or a *resuming handler* but never both, as in the left example in Figure 3. Here, the general catch clause is replaced by the specific terminate and resume clauses, which only handle exception e if it is thrown or resumed, respectively. Hence, handlers are partitioned according to the handling model. The EHMs in Exceptional C [13] and μSystem [18] have *handler partitioning*.

With handler partitioning, a handling model is determined by the time a handler is chosen, i.e., at the end of the propagation. However, the choice of handling model can be further delayed as mentioned in Section 7. The handler can decide how to handle the exception depending on the information passed to the handler or by checking the global context, as shown in the right

```
// partitioned                  // not partitioned
try { ...                       try { ...
} terminate( E e ) {            } catch( E e ) {
                                    if ( ! resumable ) {
    cleanup( e );                       cleanup( e ); unwind;
} resume( E e ) {                   } else
    recover( e );                       recover( e );
}                               }
```

Fig. 3. Handler Partitioning

example of Figure 3. The VMS and the Beta [28] exception handling systems are two examples of this approach. Obviously, more flexible handlers can be written without handler partitioning.

11 Exception Partitioning

This section discusses two issues about exception partitioning: partitioning of exceptions into categories throw-only, resume-only and dual, and implications of partitioning on derived exceptions.

11.1 Partitioning Categories

An exception can be tied to a particular propagation mechanism at declaration [2, p. 688]. A consequence is the partitioning of exceptions. An exception can be declared to be *throw-only* or *resume-only*, if it can be only thrown or only resumed respectively, or *dual* if it can be either thrown or resumed. Without partitioning, every exception becomes dual if the EHM supports throwing and resuming.

The declaration should reflect the nature of the abnormal condition causing the event being raised. For example, Unix signals SIGBUS or SIGTERM always lead to termination of an operation, and hence, should be declared as throw-only. Indeed, having throw-only and resume-only exceptions can remove the mistake of using the wrong handling or propagation mechanism.

Having dual exceptions in addition to throw-only and resume-only enhances the EHM programmability in several ways. First, encountering an abnormal condition can lead to resuming an exception or throwing one depending on the execution context. Without dual exceptions, two different exceptions must be declared, one being resume-only and the other throw-only. These two exceptions are apparently unrelated without a naming convention; using a single dual exception is simpler. Next, using a dual exception instead of resume-only for some abnormal conditions allows a resumed event to be thrown when no resuming handler is found to handle it. This effect can be achieved through a default resuming handler that throws the exception. A resume-only exception cannot be thrown. Finally, always restricting one raising mechanism to one exception has its drawbacks. Suppose a throw-only exception is declared in a library and a client of the library wants to resume it, or vice versa. The problem is that throw-only and resume-only exceptions lack the flexibility of dual, and flexibility improves reusability.

11.2 Derived Exception Implications

With derived exceptions and partitioned exceptions, there is the issue of deriving one kind from the other, say resume-only from dual, called *heterogeneous derivation*. If the derivation is restricted to exceptions of the same kind it is called *homogeneous derivation*.

Homogeneous derivation is straightforward and easy to understand. Heterogeneous derivation is complex but more flexible because it allows deriving from any kind of exception. With heterogeneous derivation, it is possible to have all exceptions in one hierarchy.

The complexity with heterogeneous derivation comes from the following heterogeneous derivations:

parent	throw	resume	dual	dual	throw	resume
derived	resume	throw	resume	throw	dual	dual
option	1		2		3	

In option 1, the kind of exception is different when the derived exception is raised and the parent is caught. If a resume-only exception is caught by a throw-only handler, it could unwind the stack, but that invalidates resumption at the raise point. If a throw-only exception is caught by a resume-only handler, it could resume the event, but that invalidates the termination at the raise point. In option 2, problems occur when the dual exception attempts to perform an unwind or resume on an exception of the wrong kind, resulting in the option 1 problems. In option 3, there is neither an obvious problem nor advantage if the dual exception is caught by the more specific parent. In most cases, it seems that heterogeneous derivation does not simplify programming and may confuse programmers; hence, it is rejected as a useful feature.

12 Matching Handling

Section 5 identifies two handling models, termination and resumption, as retrying is a special form of the termination model. Section 7 identifies two propagation mechanisms, throwing and resuming. These mechanisms are orthogonal, and hence, lead to four possible situations in an EHM, as in:

	terminating handler	resuming handler
throwing	1. matching	2. unmatching
resuming	3. unmatching	4. matching

Up to now, *matching* has been assumed between handling model and propagation mechanism, i.e., termination matches with throwing and resumption with resuming. However, the other two possibilities (options 2 and 3) must be examined to determine if there are useful semantics. In fact, this discussion parallels that for heterogeneous derivation.

In option 2, when an exception is thrown, the stack is immediately unwound and the operation cannot be resumed. Therefore, a resuming handler handling a thrown exception cannot resume the terminated operation. This semantics is misleading and difficult to understand, possibly resulting in an error long after the handler returns, because an operation throwing an exception expects a handler to provide an alternative for its guarded block, and a resuming handler catching an exception expects the operation raising it to continue. Therefore, unmatching handling of a thrown exception is largely an unsafe feature and is rejected.

In option 3, when an exception is resumed, the stack is not unwound so a terminating handler has four possibilities. First, the stack is not unwound and the exception is handled with the resumption model, i.e., the termination is ignored. Second, the stack is unwound only after the handler executes to completion. Third, the stack is unwound by executing a special statement during execution of the handler. Fourth, the stack is unwound after finding the terminating handler but before executing it. The first option is unsafe because the terminating handler does not intend to resume, and therefore, it does not correct the problem before returning to the raise point. The next two options can result in recursive resuming in the terminating handler because of outstanding handlers on the unwound stack. The problems can be avoided by the fourth option, which unwinds the stack before executing the handler, essentially handling the resumed exception as a thrown one. It also simplifies the task of writing a terminating handler because a programmer does not have to be concerned about unwinding the stack explicitly, or lexical scoping problems and side-effects from terminated blocks if the stack is unwound inside or after the terminating handler. Because of its superiority over the other two options favouring termination, the last option is the best semantics for unmatching handling of resumed exception (but is still questionable).

In matching handling, it is possible to determine what model is used to handle a raised exception (and the control flow) by knowing either how an exception is raised or which handler is chosen. Abstracting the resumption model and the termination model are done in a symmetric fashion. The same cannot be said about unmatching handling. In particular, it is impossible to tell whether a resumed exception is handled with the resumption model without knowing which handler catches it, but a thrown exception is always handled with the termination model. Hence, throwing and resuming are asymmetric in unmatching handling. Without knowing the handling model used for a resumed exception, it becomes more difficult to understand the resuming mechanism for unmatching handling than the throwing and resuming mechanism for matching handling. Therefore, unmatching handling is inferior to matching handling and is rejected.

13 HANDLER SELECTION

The propagation mechanism determines how handler clauses are searched to locate a handler. It does not specify which handler in a handler clause is chosen if there are multiple handlers capable of catching the exception. This section discusses issues about three orthogonal criteria — agreement, closeness and specificity — for choosing a handler among those capable of handling a raised exception.

13.1 Three Orthogonal Criteria

A handler clause can have several handlers capable of handling a raised event. A handler clause can handle both a derived and parent exception; it can also have a handler for an exception with and without the bound/conditional property.

The *agreement* criteria selects a handler that matches with the propagation mechanism. Agreement only applies for an EHM with the two distinct propagation mechanisms and handler partitioning.

The *closeness* criteria chooses the closest handler on the stack capable of handling an exception. A handler is closer than another if its handler clause is located prior to others based on a given propagation mechanism.

The *specificity* criteria implies an eligible handler is more specific than another (in any handler clause) if: 1) both handle the same exception and the former applies conditional handling while the other does not, or 2) the former handles an exception derived from the one handled by the latter, and both these handlers handle the exception unconditionally or conditionally (where all conditions have the same ranking). It is sometimes infeasible to tell which handler in a handler clause is more specific. In particular, a handler for an exception E and a conditional handler for an ancestor of E are equally specific.

A language designer has to set priorities among these three orthogonal criteria and there are 6 (${}_3P_3$) possible ways to order them. In addition, the priority of handling a thrown exception is orthogonal to that of a resumed one, so there are actually 36 (6^2) possible ways to define how to select a handler. Instead of looking at individual selecting schemes, it is sufficient to examine just the priorities.

13.2 Prioritizing the Criteria

Agreement should have the highest priority, when applicable, because matching handling is safe, consistent and comprehensible (see Section 12). A consequence of mandatory agreement is a terminating handler hierarchy for thrown exceptions and a resuming handler hierarchy for resumed ones. With separate *handler hierarchies*, it is reasonable for an exception to have both a default terminating handler and resuming handler (see Section 6.3 concerning default handlers). It is still possible for a default resuming handler to override resuming (see Section 7) and throw an exception in the terminating hierarchy. Overriding does not violate mandatory agreement because of the explicit throw in the handler. If there is no default handler in either case, the runtime system must take some appropriate action, usually terminating the execution.

Closeness should have the next highest priority for the reasons given in Section 8.1, i.e., handling an exception at the highest level deals with the condition in a context that is more specific, where maximum information exists.

Specificity is good, but comes after closeness to ensure a handler can protect an exception from propagating out of a guarded block. In the following, assume exception E2 is derived from E1 and specificity has higher priority than closeness:

```
try {
    try { ... throw E2; ...     // E2 is derived from E1
    } catch( E1 ) ...           // parent handler
} catch( E2 ) ...               // exact handler
```

then the handler for E2 is chosen, not the one for E1, which means E2 is propagated out of the inner guarded block. As exception derivation cannot be anticipated in general, no handler can guarantee to handle an exception and prevent it from propagating further and affecting a more abstract operation. Indeed, a library routine working perfectly if called by one client routine can behave differently if called by a different client routine because the client routine sets up its handlers differently. This semantics is a trap for programmers and potentially breaks abstraction. In addition, before committing to a particular handler, the runtime system has to search all handler clauses for an execution to ensure a more specific handler is not available.

The only exception to this strict prioritization is when two handlers in the same handler clause are equally specific for a raised exception, requiring additional criteria to resolve the ambiguity. The most common one is the position of a handler in a handler clause, e.g., select the first matching handler found in the handler-clause list. Whatever this additional criteria is, it should be applied to resolve ambiguity only after using the other three ordering criteria.

14 Preventing Recursive Resuming

Recursive resuming (see Section 8.1.3) is the only legitimate criticism of resuming propagation. Mesa probably has the only EHM that attempts to solve this problem [12, p. 143]. The rest of this section looks at the solution in Mesa and other possible solutions.

14.1 Mesa Propagation

Mesa propagation prevents recursive resuming by not reusing an unhandled handler bound to a specific called block, i.e., once a handler for a block is entered it is *marked* as unhandled and not used again. The propagation mechanism always starts from the top of the stack to find an unmarked handler for a resume exception.[1] However, this unambiguous semantics is often described as confusing.

The following program demonstrates how Mesa solves recursive resuming:

```
void test() {
    try {                                   // T1(H1(R2))
        try {                               // T2(H2(R1))
            try { resume R1;                // T3(H3(R2))
            } catch( R2 )resume R1;         // H3(R2)
        } catch( R1 ) resume R2;            // H2(R1)
    } catch( R2 ) ...                       // H1(R2)
}
```

The following stack frame is generated at the point when exception R1 is resumed from the innermost try block:

test → T1(H1(R2)) → T2(H2(R1)) → T3(H3(R2)) → H2(R1)

The potential infinite recursion occurs because H2(R1) resumes R2, and there is resuming handler H3(R2), which resumes R1, while handler H2(R1) is still on the stack. Hence, handler body H2(R1) calls handler body H3(R2) and vice versa with no case to stop the recursion.

Mesa prevents the infinite recursion by marking an unhandled handler as ineligible (in bold), resulting in:

test → T1(H1(R2)) → T2(**H2(R1)**) → T3(H3(R2)) → H2(R1)

Now, H2(R1) resumes R2, which is handled by H3(R2):

test → T1(H1(R2)) → T2(**H2(R1)**) → T3(H3(R2)) → H2(R1) → H3(R2)

Therefore, when H3(R2) resumes R1 no infinite recursion occurs as the handler for R1 in T2(H2(R1)) is marked ineligible.

However, the confusion with the Mesa semantics is that there is now no handler for R1, even though the nested try blocks appear to deal with this situation. In fact, looking at the static structure, a programmer might incorrectly assume there is an infinite recursion between handlers H2(R1) and H3(R2), as they resume one another. This confusion has resulted in a reticence by language designers to incorporate resuming facilities in new languages. In detail, the Mesa semantics has the following negative attributes:

- Resuming an exception in a block and in one of its handlers can call different handlers, even though the block and its handlers are in the same lexical scope. For instance, in the above example, an exception generated in a guarded block is handled by handlers *at* or *below* the block on the stack, but an exception generated in a handler body can be handled by handlers *above* it on the stack. Clearly, lexical scoping does not reflect the difference in semantics.
- Abstraction implies a routine should be treated as a client of routines it calls directly or indirectly, and have no access to the implementations it uses. However, if resuming from a resuming handler is a useful feature, some implementation knowledge about the handlers bound to the stack *above* it must be available to successfully understand how to make corrections, thereby violating abstraction.
- Finally, exceptions are designed for communicating abnormal conditions from callee to caller. However, resuming an exception inside a resuming handler is like abnormal condition propagating from caller to callee because of the use of handlers *above* it on the stack.

14.2 New Propagation Scheme

A new propagation mechanism for solving the recursive resuming problem, but without the Mesa problems, is presented. Further, the mechanism is extended to cover asynchronous exceptions, which Mesa does not have. Before looking at the new mechanism, the concept of consequent events is defined, which helps to explain why the semantics of the new mechanism are desirable.

14.2.1 Consequent Events

Raising an exception synchronously implies an abnormal condition has been encountered. A handler can catch an event and then raise another synchronous event if it encounters another abnormal condition, resulting in a second synchronous exception. The second event is considered a *consequent event* of the first. More precisely, every synchronous event is an *immediate consequent event* of the most recent exception being handled in the execution (if there is one). For example, in the previous Mesa resuming example, the consequence sequence is R1, R2, and R1. Therefore, a consequent event is either the immediate consequent event of an event or the immediate consequent event of another consequent event. The consequence relation is transitive, but not reflexive. Hence, synchronous events propagated when no other events are being handled are the only non-consequent events.

An asynchronous exception is not a consequent event of other exceptions propagated in the faulting execution because the condition resulting in the event is encountered by the source execution, and in general, not related to the faulting execution. Only a synchronous event raised after an asynchronous event is a consequent event of the asynchronous event.

[1] This semantics was determined with test programs and discussions with Michael Plass and Alan Freier at Xerox Parc.

14.2.2 Consequential Propagation

A new propagation mechanism is proposed, called *consequential propagation*, based on the premise that if a handler cannot handle an event, it should not handle its consequent events, either. Conceptually, the propagation searches the execution stack in the normal way to find a handler, but marks as ineligible *all* handlers inspected, including the chosen handler. Marks are cleared only when an event is handled, so any consequent event raised during handling also sees the marked handlers. Practically, all resuming handlers at each level are marked when resuming an event; however, stack unwinding eliminates the need for marking when throwing an event. Agreement eliminates the need to mark terminating handlers because only resuming handlers catch resume events. If the resuming handler overrides the propagation by throwing, the stack is unwound normally from the current handler frame.

How does consequential propagation make a difference? Given the previous Mesa runtime stack:

test → T1(H1(R2)) → T2(H2(R1)) → T3(H3(R2)) → H2(R1)

consequential propagation marks all handlers between the raise of R1 in T3(H3(R2)) to T2(H2(R1)) as ineligible (in bold):

test → T1(H1(R2)) → T2(**H2(R1)**) → T3(**H3(R2)**) → H2(R1)

Now, H2(R1) resumes R2, which is handled by H1(R2) instead of H3(R2).

test → T1(**H1(R2)**) → T2(**H2(R1)**) → T3(**H3(R2)**) → H2(R1) → H1(R2)

Like Mesa, recursive resuming is eliminated, but consequential propagation does not result in the confusing resumption of R1 from H3(R2). In general, consequential propagation eliminates recursive resuming because a resuming handler marked for a particular event cannot be called to handle its consequent events. As well, propagating a synchronously resumed event out of a handler does not call a handler bound to a stack frame between the handler and the handler body, which is similar to a thrown event propagated out of a guarded block because of stack unwinding.

Consequential propagation does not preclude all infinite recursion with respect to propagation, as in:

```
void test() {
    try { ... resume R; ...     // T(H(R))
    } catch( R ) test();        // H(R)
}
```

Here, each call of test creates a new try block to handle the next recursion, resulting in an infinite number of handlers:

test → T(**H(R)**) → H(R) → test → T(**H(R)**) → H(R) → test → ...

As a result, there is always an eligible handler to catch the next event in the recursion. Consequential propagation is not supposed to handle this situation as it is considered an error with respect to recursion not propagation.

Finally, consequential propagation does not affect throwing propagation, because marked resuming handlers are simply removed during stack unwinding. Hence, the application of consequential propagation is consistent with either throwing or resuming. As well, because of partitioning, a terminating handler for the same event bound to a prior block of a resuming handler is still eligible, as in:

```
void test() {
    try { ... resume R; ...     // T(r(R),t(R))
    } terminate( R ) ...        // t(R)
      resume( R ) throw R;      // r(R)
}
```

Here, the resume of R in the try block is handled by the specific handler resume(R), which then throws the exception and it is handled by the specific handler terminate(R), resulting in the following call stack:

test → T(**r(R)**,t(R)) → r(R) → t(R)

Notice, the resuming handler for R is marked ineligible during the resume of R, and the terminating handler for the same try block is still eligible to handle the throw of R.

All handlers are considered unmarked for a propagated asynchronous event because an asynchronous event is not a consequent event. Therefore, the propagation mechanism searches every handler on the runtime stack. Hence, a handler ineligible to handle an event and its consequent events can be chosen to handle a newly arrived asynchronous event, reflecting its lack of consequentiality.

In summation, consequential propagation is better than other existing propagation mechanisms because:

- it supports throwing and resuming propagation, and the search for a handler occurs in a uniformly defined way,
- it prevents recursive resuming and handles synchronous and asynchronous exceptions according to a sensible consequence relation among exceptions,
- the context of a handler closely resembles its guarded block with respect to lexical location; in effect, an event propagated out of a handler is handled as if the event is directly propagated out of its guarded block.

15 MULTIPLE EXECUTIONS AND THREADS

The presence of multiple executions and multiple threads has an impact on an EHM. In particular, each execution has its own stack on which threads execute, and the different threads can carry out the various operations associated with handling a single exception. For example, the thread of the source execution raises an exception in the faulting execution executed by another thread, which finally propagates and handles it.

15.1 Coroutine Environment

Coroutining represent the simplest execution environment where the source execution can be different from the faulting execution, but the thread of a single task executes both source and faulting execution. In theory, either execution can propagate the event, but in practice, only the faulting execution is reasonable. Assume the source execution propagates the event in the following:

```
try {                           // T1(H1(E1))
    try { EX1 (suspended)       // T2(H2(E2))
    } catch( E2 ) ...           // H1(E2)
} catch( E1 ) ...               // H2(E1)
```

and execution EX1 is suspended in the guarded region T2. While suspended, a source execution EX2 raises and propagates an asynchronous exception E1 in EX1, which directs control flow of EX1 to handler H2(E1), unwinding the stack in the process. While EX1 is still suspended, a third source execution EX3 raises and propagates another asynchronous exception E2 (EX2 and EX3 do not have to be distinct). Hence, control flow of EX1 goes to another handler determined in the dynamic context, further unwinding the stack. The net effect is that neither of the exceptions is handled by any handler in the program fragment.

The alternative approach is for the faulting execution, EX1, to propagate the exceptions. Regardless of which order EX1 raises the two arriving events, at least a handler for one of the events is called. Because of the potential for confusion, only the faulting execution should propagate an exception in an environment with multiple executions.

15.2 Concurrent Environment

Concurrency represents the complex execution environment where the separate source and faulting executions are executed by the threads of different tasks. In theory, either execution can propagate the event, but in practice, only the faulting execution is reasonable. If the source execution propagates the event, it must change the faulting execution, including the runtime stack and program counter. Consequently, the runtime stack and the program counter become shared resources between tasks, making a task's execution dependent as other task's execution in a direct way, i.e., not through communication. To avoid corrupting an execution, locking is now required. Hence, an execution must lock its own runtime stack before execution and blocking, respectively. Obviously, this approach generates a large amount of superfluous lockings to deal with a situation that occurs rarely. Therefore, it is reasonable to allow only the faulting execution to propagate an exception in an environment with multiple tasks.

15.3 Real-Time Environment

In the design and implementation of real-time programs, various timing constraints are guaranteed through the use of scheduling algorithms, as well as an EHM. Exceptions are extremely crucial in real-time systems, e.g, deadline expiry or early/late starting exceptions, as they allow a system to react to abnormal situations in a timely fashion. Hecht [29] demonstrated, through various empirical studies, that the introduction of even the most basic fault-tolerance mechanisms into a real-time system drastically improves its reliability.

The main conflict between real-time and an EHM is the need for constant-time operations and the dynamic choice of a handler [30]. As pointed out in Section 8.1.2, the dynamic choice of a handler is crucial to an EHM, and therefore, it may be impossible to resolve this conflict. At best, exceptions may only be used in restricted ways in real-time systems when a bound can be established on call stack depth and the number of active handlers, which indirectly puts a bound on propagation.

16 Asynchronous Events

The protocol for communicating asynchronous events among coroutines and tasks is examined.

16.1 Communication

Because only the faulting execution should propagate an event and directly alter control flow, the source must inform the faulting execution to propagate an event. This requires a form of direct communication not involving any shared object. In essence, an event is transmitted from the source to the faulting execution.

There are two major categories of direct communication: blocking and non-blocking. In the first, the sender blocks until the receiver is ready to receive the event; in the second, the sender does not block. In both cases, the receiver is blocked if the sender has yet to send an event.

16.1.1 Source Execution Requirement

Using blocking communication, the source blocks until the faulting execution executes a complementary receive. However, an execution may infrequently (or never) check for incoming events. Hence, the source can be blocked for an extended period of time waiting for the faulting execution to receive the exception event. Therefore, blocking communication is rejected. Only non-blocking communication allows the source to raise an exception on one or more executions without suffering an extended delay.

16.1.2 Faulting Execution Requirement

Non-blocking communication for exceptions is different from ordinary non-blocking communication. In the latter case, a message is delivered only after the receiver executes some form of receive. The former requires the receiver to receive an exception event without explicitly executing a receive because an EHM should preclude checking for an abnormal condition. The programmer is required to set up a handler only to handle the rare condition. From the programmer's perspective, the delivery of an asynchronous exception is transparent. Therefore, the underlying system must poll for the arrival of such an exception, and propagate it on arrival. The delivery of asynchronous exceptions must be timely, but not necessarily immediate.

There are two polling strategies: *explicit polling* and *implicit polling*. Explicit polling requires the programmer to insert explicit code to activate the polling. Implicit polling is performed by the underlying system. (Hardware interrupts involve implicit polling because the CPU automatically polls for the event.)

Explicit polling gives a programmer control over when an asynchronous exception can be raised. Therefore, the programmer can delay, or even completely ignore pending asynchronous exceptions. Delaying and ignoring asynchronous exceptions are both undesirable. The other drawback of explicit polling is that a programmer has to worry about when to and when not to poll, which is equivalent to explicitly checking for exceptions.

Implicit polling alleviates programmers from polling, and hence, provides an apparently easier interface to programmers. On the other hand, implicit polling has its own drawbacks. First, infrequent implicit polling can delay the handling of asynchronous exceptions; polling too frequently can deteriorate the runtime efficiency. Without specific knowledge of a program, it is difficult to have the right frequency for implicit polling. Second, implicit polling suffers the non-reentrant problem (discussed next).

Unfortunately, an EHM with asynchronous exceptions needs to employ both implicit and explicit polling. Implicit polling simplifies using the EHM and reduces the damage a programmer can do by ignoring asynchronous exceptions. However, the frequency of implicit polling should be low to avoid unnecessary loss of efficiency. Explicit polling allows programmers to have additional polling when it is necessary. The combination of implicit and explicit polling gives a balance between programmability and efficiency. Finally, certain situations require any implicit polling to be turned off, possibly by a compiler or runtime switch, e.g., low-level system code where execution efficiency is crucial or real-time programming to ensure deadlines.

16.2 Non-Reentrant Problem

Asynchronous events introduce a form of concurrency into sequential execution because delivery is non-deterministic with implicit polling. The event delivery can be thought as temporarily stealing a thread to execute the handler. As a result, it is possible for a computation to be interrupted while in an inconsistent state, a handler found, and the handler recursively calls the inconsistent computation, called the *non-reentrant problem*. For example, while allocating memory, an execution is suspended by delivery of an asynchronous event, and the handler for the exception attempts to allocate memory. The recursive entry of the memory allocator may corrupt its data structures.

The non-reentrant problem cannot be solved by locking the computation, because either the recursive call deadlocks on the second call or for recursive locks, reenters and corrupts the data. To ensure the correctness of a non-reentrant routine, the execution must block the delivery, and consequently the propagation, of asynchronous exceptions, hence temporarily precluding delivery.

Hardware interrupts are also implicitly polled by the CPU. The non-reentrant problem can occur if the interrupt handler enables the interrupt and recursively calls the same computation as has been interrupted. However, because hardware interrupts can happen at times when asynchronous exceptions cannot, it is more difficult to control delivery.

16.3 Disabling Asynchronous Exceptions

Because of the non-reentrant problem, facilities must exist to disable asynchronous exceptions. There are two aspects to disabling: the specific event to be disabled and the duration of disabling. (This discussion is also applicable to hardware interrupts and interrupt handlers.)

16.3.1 Specific Event

Without derived exceptions, only the specified exception is disabled; with derived exceptions, the exception and all its descendants can be disabled. Disabling an individual exception but not its descendents, called *individual disabling*, is tedious as a programmer must list all the exceptions being disabled, nor does it complement the exception hierarchy. If a new derived exception should be treated as an instance of its ancestors, the exception must be disabled wherever its ancestor is disabled. Individual disabling does not automatically disable the descendents of the specified exceptions, and therefore, introducing a new derived exception requires modifying existing code to prevent it from activating a handler bound to its ancestor. The alternative, *hierarchical disabling*, disables an exception and its descendents. The derivation becomes more restrictive because a derived exception also inherits the disabling characteristics of its parent. Compared to individual disabling, hierarchical disabling is more complex to implement and usually has a higher runtime cost. However, the improvement in programmability makes hierarchical disabling attractive.

A different approach is to use priorities instead of hierarchical disabling, allowing a derived exception to override its parent's priority when necessary. Selective disabling can be achieved by disabling exceptions of priority lower than or equal to a specified value. This selective disabling scheme trades off the programmability and extensibility of hierarchical disabling for lower implementation and runtime costs. However, the problem with priorities is assigning priority values. Introducing a new exception requires an understanding of its abnormal nature plus its priority compared to other exceptions. Hence, defining a new exception requires an extensive knowledge of the whole system with respect to priorities, which makes the system less maintainable and understandable. It is conceivable to combine priorities with hierarchical disabling; a programmer specifies both an exception and a priority to disable an asynchronous exception. However, the problem of maintaining consistent priorities throughout the exception hierarchy still exists. In general, priorities are an additional concept that increases the complexity of the overall system and are rejected.

Therefore, hierarchical disabling with derived exceptions seems the best approach in an extensible EHM. Note that multiple derivation (see Section 6.2) only complicates hierarchical disabling, and the same arguments can be used against hierarchical disabling with multiple derivation.

16.3.2 Duration

The duration for disabling could be specified by a time duration, but normally the disabling duration is specified by a region of code that cannot be interrupted. There are several mechanisms available for specifying the region of uninterruptable code.

One approach is to supply explicit routines to turn on and off the disabling for particular asynchronous exceptions. However, the resulting programming style is like using a semaphore for locking and unlocking, which is a low-level abstraction. Programming errors result from forgetting a complementary call and are difficult to debug.

An alternative is a new kind of block, called a *protected block*, which specifies a list of asynchronous events to be disabled and the associated region of code. On entering a protected block, the listed of disabled asynchronous events is modified, and subsequently enabled when the block exits. The effect is like entering a guarded block.

An approach suggested for Java [31] associates the disabling semantics with an exception named AIE. If a member routine includes this exception in its exception list, interrupts are disabled during execution of the member; hence, the member body is the protected block. However, this approach is poor language design because it associates important semantics with a name, AIE, and makes this name a hidden keyword.

The protected block seems the simplest and most consistent in an imperative language with nested blocks. Regardless of how asynchronous exceptions are disabled, all events (except for special system events) should be disabled initially for an execution; otherwise an execution cannot install handlers before asynchronous events begin arriving.

16.4 Multiple Pending Asynchronous Exceptions

Since asynchronous events are not serviced immediately, there is the potential for multiple events to arrive between two polls for events. There are several options for dealing with these pending asynchronous events.

Asynchronous events do not have to be queued, so each execution has a buffer for only one pending event. New events would be are discarded after the first one arrives, or overwritten as new ones arrive, or overwritten only for higher priority events. However, the risk of losing an asynchronous event makes a system less robust; hence queuing events is usually superior.

For queueing, the options are the service order of the list. The order of arrival can be chosen to determine the service order of handling pending events. However, a strict FIFO delivery order may be unacceptable, e.g., an asynchronous event to stop an execution from continuing erroneous computation can be delayed for an extended period of time in a FIFO queue. A more flexible semantics for handling pending exceptions is user-defined priorities. However, Section 16.3 discusses how a priority scheme reduces extensibility, making it inappropriate in an environment emphasizing code reuse.

Therefore, FIFO order seems acceptable for its simplicity in understanding and low implementation cost. However, allowing a pending event whose delivery is disabled to prevent delivering other pending events seems undesirable. Hence, an event should be able to be delivered before earlier events if the earlier events are disabled. This out of order delivery has important implications on the programming model of asynchronous exceptions. A programmer must be aware of the fact that two exceptions having the same source and faulting execution may be delivered out of order (when the first is disabled but not the second). This approach may seem unreasonable, especially when causal ordering is proved to be beneficial in distributed programming. However, out of order delivery is necessary for urgent events. Currently, the most adequate delivery scheme remains as an open problem, and the answer may only come with experience.

16.5 Converting Interrupts to Exceptions

As mentioned, hardware interrupts can occur at any time, which significantly complicates the non-reentrant problem. One technique that mitigates the problem is to convert interrupts into language-level asynchronous events, which are then controlled by the runtime system. Some interrupts target the whole program, like abort execution, while some target individual executions that compose a program, like completion of a specific thread's I/O operation. Each interrupt handler raises an appropriate asynchronous exception to the particular faulting execution or to some system execution for program faults. However, interrupts must still be disabled when enqueueing and dequeuing the asynchronous events to avoid the possibility of corrupting the queue by another interrupt or the execution processing the asynchronous events. By delivering interrupts through the EHM, the non-reentrant problem is avoided and interrupts disable for minimal time. Furthermore, interrupts do not usually have all the capabilities of an EHM, such as parameters; hence, interrupts are not a substitute for a general EHM. Finally, the conversion also simplifies the interface within the language. The interrupts can be completely hidden within the EHM, and programmers only need to handle abnormal conditions at the language level, which improves portability across systems. However, for hard real-time systems, it may still be necessary to have some control over interrupts as they can invalidate timing constraints.

One final point about programming interrupt handlers is that raising a synchronous exception within an interrupt handler is meaningful only if it does not propagate outside of the handler. The reason is that the handler executes on an arbitrary execution stack, and hence, there is usually no relationship between the interrupt handler and the execution. Indeed, Ada 95 specifies that propagating a thrown event from an interrupt handler has no effect.

17 Conclusions

Raising, propagating and handling an exception are the three core control-flow mechanisms of an EHM. For safety, an EHM should provide two raising mechanisms: throwing and resuming. There are two useful handling models: termination and resumption. Handlers should be partitioned with respect to the handling models to provide a better abstraction. Exception parameters, homogeneous derivation of exceptions and bound/conditional handling all improve programmability and extensibility. In a concurrent environment, an EHM must provide some disabling facilities to solve the non-reentrant problem. Hierarchical disabling is best in terms of programmability and extensibility. The new propagation mechanism proposed, consequential propagation, solves the recursive resuming problem and provides consistent propagation semantics, making it better than existing propagation mechanisms. As a result, the resumption model becomes attractive and can be introduced into existing termination-only EHMs. An EHM based on the ideas presented in this paper has been implemented in μC++ [19], providing feedback on correctness. We plan to report on practical experience in another paper.

Acknowledgments

We would like to acknowledge the suggestions made by Bob Zarnke, Ric Holt, and referee #1.

References

[1] P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke, "μC++: Concurrency in the object-oriented language C++," *Software—Practice and Experience*, vol. 22, no. 2, pp. 137–172, Feb. 1992.

[2] J. B. Goodenough, "Exception handling: Issues and a proposed notation," *Communications of the ACM*, vol. 18, no. 12, pp. 683–696, Dec. 1975.

[3] Intermetrics, Inc., *Information technology — Programming languages — Ada*, international standard ISO/IEC 8652:1995(E) edition, Dec. 1994, Language and Standards Libraries, V 6.0.

[4] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson, "Modula-3 report," Tech. Rep. 31, Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, Aug. 1988.

[5] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, third edition, 1997.

[6] Shaula Yemini and Daniel M. Berry, "A modular verifiable exception-handling mechanism," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 2, pp. 214–243, Apr. 1985.

[7] International Business Machines, *OS and DOS PL/I Reference Manual*, first edition, Sept. 1981, Manual GC26-3977-0.

[8] M. Donald MacLaren, "Exception handling in PL/I," *SIGPLAN Notices*, vol. 12, no. 3, pp. 101–104, Mar. 1977, Proceedings of an ACM Conference on Language Design for Reliable Software, March 28–30, 1977, Raleigh, North Carolina, U.S.A.

[9] Peter A. Buhr, "Are safe concurrency libraries possible?," *Communications of the ACM*, vol. 38, no. 2, pp. 117–120, Feb. 1995.

[10] Robin Milner and Mads Tofte, *Commentary on Standard ML*, MIT Press, Cambridge, Massachusetts, U.S.A., 1991.

[11] James Gosling, Bill Joy, and Guy Steele, *The Java Language Specification*, Addison-Wesley, 1996.

[12] James G. Mitchell, William Maybury, and Richard Sweet, "Mesa language manual," Tech. Rep. CSL–79–3, Xerox Palo Alto Research Center, Apr. 1979.

[13] N. H. Gehani, "Exceptional C or C with Exceptions," *Software—Practice and Experience*, vol. 22, no. 10, pp. 827–848, Oct. 1992.

[14] Bertrand Meyer, *Eiffel: The Language*, Prentice Hall Object-Oriented Series. Prentice-Hall, 1992.

[15] Steven J. Drew and K. John Gough, "Exception handling: Expecting the unexpected," *Computer Languages*, vol. 20, no. 2, May 1994.

[16] Barbara H. Liskov and Alan Snyder, "Exception handling in CLU," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 6, pp. 546–558, Nov. 1979.

[17] Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994.

[18] Peter A. Buhr, Hamish I. Macdonald, and C. Robert Zarnke, "Synchronous and asynchronous handling of abnormal events in the μSystem," *Software—Practice and Experience*, vol. 22, no. 9, pp. 735–776, Sept. 1992.

[19] Peter A. Buhr and Richard A. Stroobosscher, "μC++ annotated reference manual, version 4.8," Tech. Rep., Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, Jan. 2001, ftp://plg.uwaterloo.ca/pub/uSystem/uC++.ps.gz.

[20] Andrew Koenig and Bjarne Stroustrup, "Exception handling in C++," *Journal of Object-Oriented Programming*, vol. 3, no. 2, pp. 16–33, July/August 1990.

[21] T. A. Cargill, "Does C++ really need multiple inheritance?," in *USENIX C++ Conference Proceedings*, San Francisco, California, U.S.A., Apr. 1990, USENIX Association, pp. 315–323.

[22] Wing Yeung Russell Mok, "Concurrent abnormal event handling mechanisms," M.S. thesis, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, Sept. 1997, ftp://plg.uwaterloo.ca/pub/uSystem/MokThesis.ps.gz.

[23] Lawrence J. Kenah, Ruth E. Goldenberg, and Simon F. Bate, *VAX/VMS Internals and Data Structures Version 4.4*, Digital Press, 1988.

[24] Jørgen Lindskov Knudsen, "Exception handling — a static approach," *Software—Practice and Experience*, vol. 14, no. 5, pp. 429–449, May 1984.

[25] Jørgen Lindskov Knudsen, "Better exception handling in block structured systems," *IEEE Software*, vol. 4, no. 3, pp. 40–49, May 1987.

[26] G. Motet, A. Mapinard, and J. C. Geoffroy, *Design of Dependable Ada Software*, Prentice Hall, 1996.

[27] R. D. Tennent, "Language design methods based on semantic principles," *Acta Infomatica*, vol. 8, no. 2, pp. 97–112, 1977, reprinted in [32].

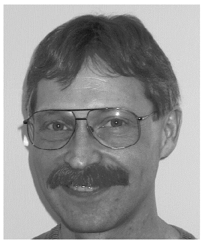
[28] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard, *Object-oriented Programming in the BETA Programming Language*, Addison-Wesley, 1993.

[29] H. Hecht and M. Hecht, "Software reliability in the systems context," *IEEE Transactions on Software Engineering*, vol. 12, no. 1, pp. 51–58, 1986.

[30] Jun Lang and David B. Stewart, "A study of the applicability of existing exception-handling techniques to component-based real-time software technology," *ACM Transactions on Programming Languages and Systems*, vol. 20, no. 2, pp. 274–301, Mar. 1998.

[31] Real Time for Java Experts Group, http://www.rtj.org, Sept. 1999.

[32] Anthony I. Wasserman, Ed., *Tutorial: Programming Language Design*, Computer Society Press, 1980.



Peter A. Buhr received the BSc Hons/MSc and PhD degrees in computer science from the University of Manitoba in 1976, 1978, 1985, respectively. He is currently an Associate Professor in the Department of Computer Science, University of Waterloo, Canada. His research interests include concurrency, concurrent profiling/debugging, persistence and polymorphism. Dr. Buhr is a member of the Association of Computing Machinery.

W.Y. Russell Mok received the BCompSc and MMath degrees in computer science from Concordia University (Montreal) in 1994 and University of Waterloo in 1998, respectively. He is currently working at Algorithmics, Toronto. His research interests include object-oriented design, software patterns and software engineering.