Concurrent Abnormal Event Handling Mechanisms

by

Wing Yeung Russell Mok

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 1997

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

This thesis is about concurrent abnormal event (or exception) handling mechanisms (AEHM's). An AEHM is a crucial programming feature for building robust, reliable software, giving programmers the ability to deal with a wide variety of dynamic events. While many modern concurrent programming languages provide some form of AEHM, the interaction between the AEHM and the concurrency features is weak; in essence, the AEHM is only applicable for sequential programming.

A framework for a comprehensive, easy to use, and extensible AEHM is given. The pros and cons of various features are studied, and new applications of features like derived events and bound events are presented. Based on the framework, previous criticisms against the resume mechanism — a feature for handling abnormal events commonly missing in modern languages — are properly addressed. Indeed, successfully incorporating the resume mechanism is a significant, though not radical, improvement in providing a comprehensive AEHM.

An AEHM is implemented in μC++ based on the framework. It supports resumption and termination, two models for handling abnormal events. It has synchronous and asynchronous events, derived events, and hierarchical blocking of asynchronous events.

μC++ is a concurrent object-oriented language with an extensive concurrent programming facilities including mutual exclusion, synchronization and context swtiching. These facilities lead to abnormal conditions normally not encountered in other languages. Nonetheless, these conditions are properly handled using the new features provided by the μC++ AEHM.

Acknowledgements

There are numerous people I have to be thankful for and my family has to be the first on the list, especially my parents and my younger sister. Thanks for putting up with me for so many years, and giving me the opportunity and freedom to pursue the education I want.

I also want to show my gratitude to my supervisor Peter Buhr for giving me the guidance both on and off campus as well as being understanding and supportive, and Robert Zarnke for being involved in the early stage of my work. Ric Holt has also given me valuable advice in making this thesis better.

Special thanks go to Philip and Kai for everything that I have learnt from you guys; Daniel Fong, William and Donna for your generosity and hospitality; and Daniel Tsui for telling me things from religion and philosophy to options, futures and epidemiology. Last but not least, I want to thank Kimmy, Konrad, Ronnie, Vernon, Ming and Winnie for being a good friend of mine in the past year.

Contents

1 Background and Motivation **1**

1.1 Motivation 1

1.2 Characteristics of an AEHM 3

1.3 Structure of the execution environment 5

1.4 Thesis outline 9

2 Framework of AEHM **10**

2.1 Overview of an AEHM 11

2.2 AEHM in a language with nested blocks 13

2.3 Stack unwinding 26

2.4 Existing propagation models 30

2.5 Context of a handler 38

2.6 Design options in handlers 42

2.7 Matching and unmatching handling 43

2.8 Selecting a handler 46

2.9 Concurrency and handling abnormal events 52

2.10 Issues specific to asynchronous events 55

2.11 Preventing Recursive Signalling 69
2.12 Abnormal event name space 77
2.13 Event parameters in derived events 80
2.14 Summary 82

3 Using abnormal events in μC++ 83

3.1 Overview of μC++ AEHM 83
3.2 Defining abnormal Events 85
3.3 Raising an abnormal event 88
3.4 Handlers 90
3.5 Delivery of asynchronous events 93
3.6 Summary 95

4 Programming with μC++ abnormal events 96

4.1 General guide lines 97
4.2 Synchronized operation 100
4.3 μC++ rendezvous 102
4.4 Abnormal conditions in the μC++ kernel 107
4.5 Abnormal events as a flow control mechanism 113
4.6 Summary 115

5 Conclusion and future work 120

A Glossary 122

Bibliography 126

List of Tables

1.1 Fundamental Concurrency Abstractions 7

2.1 Summary of the propagation options with regards to stack unwinding 30

2.2 Matching and unmatching handling 46

List of Figures

1.1 The effect of `resume` and `suspend` in coroutine 7

2.1 Subtle differences between two simple retrying handlers 16

2.2 An example of bound event 22

2.3 Handlers overriding default propagation mechanism 27

2.4 Effect of overriding throwing by signalling 27

2.5 Static choice of handler 32

2.6 An example of recursive signal 34

2.7 Drawbacks of static propagation 37

2.8 Handlers in Ada and C++ : identical semantics 39

2.9 Using resumption model to handle a signalled event 40

2.10 Static scoping in a resuming handler 41

2.11 A handler that can terminate and resume 42

2.12 Example showing the order of priority 49

2.13 Defining default handlers in C++ 52

2.14 Possible undesirable behaviour in coroutine environment 54

2.15 Abnormal events in an environment with multiple executions 55

2.16 Preventing thrown events from escaping an interrupt service routine 67

2.17 Handler marking in Mesa 71
2.18 Marking handlers 73
4.1 Synchronization 100
4.2 An example of `uWait` and `uSignal` 104
4.3 Ownership of a mutex object when executing an accepted entry . . 106
4.4 Entering a mutex object multiple times 106
4.5 μC++ abnormal event hierarchy 112
4.6 A maze search program using signalled abnormal events 114
4.7 Sather style iterator in μC++ 116
4.8 μC++ implementation of an inorder tree iterator 118

Chapter 1

Background and Motivation

Substantial research has been done on abnormal events[1][2] but there is hardly any agreement on what an abnormal event is. Attempts have been made to define abnormal events in terms of errors but an error itself is also ill-defined. Instead of defining what an abnormal event is, I treat an Abnormal Event Handling Mechanism(AEHM) as a flow control mechanism and an abnormal event is a component of the mechanism. The motivation for introducing a new flow control mechanism is to make certain programming tasks easier, in particular, writing robust programs.

1.1 Motivation

Some conditions, like run-time stack overflow, happen only rarely during execution but have to be handled to avoid abrupt termination of or damage to a system. Frequently, the occurrence of a rare (or abnormal) condition is considered as an

[1]The term exception in [2] is different from abnormal event, but in most of the literature, exception is usually used to refer to abnormal events.

error, but not always. Before abnormal event handling facilities[8], the common programming techniques used to handle these abnormal events were *status return values* and *status flags*.

The status return value 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. Instead of using special return values to indicate the occurrence of a rare condition, or in conjunction with, a shared variable is used as a status flag. Setting a flag indicates a rare condition has occurred. The value stays in the flag as long as it is not overwritten.

These two techniques have noticeable drawbacks. Both techniques require a programmer to make explicit tests — testing the return value from a routine or a status flag. These tests are located throughout the program, which reduce readability and maintainability, making it difficult to see if all the necessary error cases are covered, and inconvenient to use as a routine can encounter many different errors. As a result, testing a return value or various status flags can be significantly complicated.

In addition, both techniques are inflexible in accommodating changes that lead to new abnormal events as the checking code is written inline. Making new abnormal events compatible with existing code written before the new events are defined is difficult. The overall extensibility of a system is hence limited. It is impossible to change a value representing an abnormal event into a normal return value, or vice versa because such changes affect existing libraries. Furthermore, the techniques of status return values and status flags do not complement others flexible programming styles, like subtyping and call-back routines.

The status return value technique has another drawback. It may encode different abnormal events among normal returned values. This approach artificially enlarges the subrange of valid values independent of the computation. It becomes a programmer's responsibility not to use an error value returned from a library function as a normal value.

The most significant drawback of using status flags is that an error is only discovered and subsequently handled when the execution checks the flag. Without timely checking of the flag, 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. Another drawback is the side effects in setting flags, which make a program less understandable.

1.2 Characteristics of an AEHM

The drawbacks of the status return value and the status flag techniques indicate that additional language features are necessary for programming that deals with abnormal events. An AEHM must:

- alleviate the programmer from testing for the occurrence of rare conditions throughout the program, and from explicitly changing the control flow of the program,
- be extensible so that new abnormal events can be easily introduced,
- provide a mechanism to prevent the current incomplete operation from continuing.

The first item targets readability and programmability by removing the code written for checking status return values or status flags. The second item targets extensibility in two ways. First, a programmer can introduce new abnormal events, and second, new abnormal events should have minimal effects on existing libraries. The last item provides a transfer control mechanism from the point of the abnormal event that disallows control returning to this point. A possible use is to direct control flow away from a place where local information are corrupted. An operation that is unsafe to continue after an abnormal condition occurs is described as **non-resumable**.

Two examples of AEHM's are given to illustrate these three characteristics.

1.2.1 Unix signal mechanism

The Unix signal mechanism can be considered as an abnormal event handling mechanism. On encountering a rare condition, a signal is generated, which causes an execution to be preempted and a routine to be invoked. The transfer of control flow to the routine is different from other routine invocations because the programmer does not make the call explicitly. The operation before the signal occurs is suspended. Normally, this operation resumes after executing the routine, but with a non-local jump facility, the routine can terminate the suspended operation. A non-local jump transfers control among dynamic blocks. The destination (block) must have been invoked prior to the current block. Blocks invoked after the destination are terminated prior to restoring the destination's context. Without a non-local jump, the termination is infeasible, and hence, the Unix signal mechanism should not be considered as an AEHM[2]. Extensibility is quite limited though, as only a few signals are not predefined and available to programmers. If a library has to use one of the user available signals in Unix, all its clients must agree on this signal's definition, which makes the library less reusable.

1.2.2 Ada exception mechanism

An Ada programmer can declare a new exception as long as there is no name conflict. Hence the mechanism is extensible. When an execution encounters 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 explicitly. The operation encountering the rare condition can never be resumed, and possibly multiple active blocks are also terminated between the raise point and the handling statements.

1.3 Structure of the execution environment

The structure of the execution environment has a significant effect on the abnormal event handling mechanism, for example, an execution environment with concurrency requires a more complex AEHM than a sequential environment. I adopt the execution model described in [1], which identifies three elementary execution properties:

[2]Similarly, without label variables and the non-local `goto` statement, the `on` condition in `PL/I` cannot be considered as an AEHM either. Non-local jump in `PL/I` is discussed in section 2.2.1.

execution state — The local context information of a sequential computation is an execution state. It includes local and global data, current execution location, and routine activation records. An *execution* is an object implementing an sequential computation. Therefore, its object state is the execution state. An execution is *active* when it is bound to a thread, and is *inactive* otherwise.

thread — A thread sequentially executes programming language statements, independently of and possibly concurrently with other threads. Sequential computation is carried out as a thread changes execution states. A thread is *blocked* when it is waiting for some event to occur; it is *running* when it is executing on a processor where a processor is any unit that can carry out a computation[3]; and it is *ready* when it is neither blocked or running.

mutual exclusion — Mutual exclusion permits an operation on a shared resource to be performed without interruption by other operations on the resource.

Two threads are running in *parallel* if they are running on two different processors. Two executions are *concurrent* if they are bound to different threads. Any computation requires binding an execution (state) to a thread. *Context switch* refers to a thread switching from one execution to another, i.e. a change in the execution/thread binding[4]. Table 1.1 shows objects found in a concurrent environment and their fundamental properties. AEHM affects the control flow of an execution,

[3]To avoid any confusion, a thread can be thought as a user thread while a processor as a kernel thread.

[4]The underlying system can change the binding between threads and processors but not between executions and threads.

Table 1.1: Fundamental Concurrency Abstractions

object properties		object's member routine properties	
thread	execution state	no implicit mutual exclusion	implicit mutual exclusion
no	no	1 class-object	2 monitor
no	yes	3 coroutine	4 coroutine-monitor
yes	no	5 N.A.	6 N.A.
yes	yes	7 N.A.	8 task

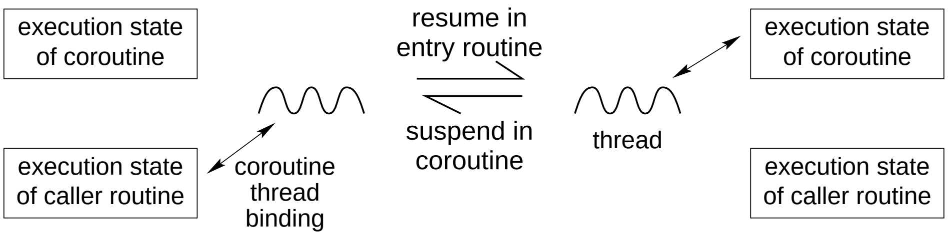


Figure 1.1: The effect of `resume` and `suspend` in coroutine

and it is often sufficient when discussing abnormal events to refer to an execution without knowing if it is a coroutine, a coroutine-monitor, or a task. Furthermore, coroutine-monitor is often treated as a coroutine in situation where the thread aspect is important. For an in depth discussion on the concurrency abstractions in the model, consult the original paper.

A *coroutine* has an execution state but no thread. It becomes active when it is bound to a thread; otherwise, it is inactive. The coroutine-thread binding is a many-to-one relation in the execution model. The execution state of a coroutine can be resumed and suspended, causing a context switch to and from the execution of the coroutines. Figure 1.1 shows the change of execution/thread binding as a result of calling the coroutine operations `resume` and `suspend`. Each coroutine has a set of *entry routines*, which are invoked[5] by other executions. The execution, which can be a task or another coroutine, invoking an entry routine creates a stack frame on its own stack for the entry routine. Executing a `resume` statement in the entry routine causes a context switch to the coroutine. The first execution that resumes a coroutine *starts* the coroutine. Executing a `suspend` statement causes another context switch, which suspends the coroutine's execution and resumes the execution that performed the previous `resume`.

The coroutine can be thought of as an abstraction of a finite state machine and can be used to achieve the effect of lazy generation of an infinite list of values. Since coroutines binding to the same thread are not concurrent, there is no race condition among them. I call the single-threaded environment with multiple coroutines the *coroutine environment.*

A *task* has an execution state and a thread and is the only entity in the execution model that has a thread, meaning that a thread must be created and destroyed together with a task. Therefore, the mapping from task to thread is one-to-one. While a task owns the thread, it can relinquish its thread to a coroutine by context switching to the coroutine using `resume`, consequently changing the thread-execution binding.

An environment having at least two tasks is a *concurrent environment.* In other words, a concurrent environment has multiple threads and multiple executions.

[5] In this thesis, the word "call" refers to a statement. "Invoke" and "activate" are reserved for the execution of a call statement.

1.4 Thesis outline

Chapter 2 establishes a framework for an AEHM. The various components of an AEHM are studied and discussed. Components are examined independently, if appropriate, before looking at how they interact. A comprehensive design for a concurrent AEHM is proposed in this chapter as well.

Chapter 3 is about the μC++ AEHM that I implemented based on the proposed design. It describes the available features and the syntax.

Chapter 4 highlights the use of abnormal events in μC++ kernel. It describes the abnormal conditions that may arise at runtime. Previously, these conditions either cause the program to abort immediately, or are not detected until resulting in other errors, depending on the compiler flags used[6]. Abnormal events introduced to allow detection of and recovery from these conditions are described in this chapter. Additional examples of using abnormal events in μC++ are also given.

Chapter 5 is the conclusion with future work, followed by a terminology glossary in the appendix for easy reference during reading.

[6]Runtime checks for certain abnormal conditions are turned on with the `-debug` flag.

Chapter 2

Framework of AEHM

This chapter presents a comprehensive analysis on an AEHM. Section 2.1 is an overview of an AEHM as a flow control mechanism[1]. It outlines the steps in the change of flow control for an AEHM, namely *raising* an event, *propagating* an event and *handling* an event. The handling step is further discussed in section 2.2, along with other programming features available in previously implemented AEHM's. Sections 2.3 and 2.4 are about propagating an event. In particular, a survey on different propagating mechanisms is presented. The recursive signalling problem is also introduced. Section 2.5 looks at how propagation may affect the context in which a handler executes, where a handler is a sequence of statements written to handle abnormal events. Section 2.6 describes different designs for declaring and defining a handler.

Sections 2.7 and 2.8 deal with issues that arise when integrating the pieces mentioned above. The discussion shows that different features in propagation and

[1]Loop and function call statements are other examples of flow control mechanism.

handlers do not always complement each other.

Section 2.9 focuses on how a sequential AEHM with a single execution may be extended to an environment of multiple executions, followed by a discussion on using asynchronous abnormal events and programming features making asynchronous abnormal events safe to use. I also introduce *hierarchical blocking* of asynchronous abnormal events in this section 2.10.3.

Section 2.11 discusses solutions to the recursive signalling problem. At the end, I propose a new propagation mechanism that eliminates the problem and supports the use of asynchronous abnormal events.

The next section looks at how abnormal events can be partitioned and the use of an abnormal event hierarchy, i.e., features related to the abnormal event name space structure. The event hierarchy must be used cautiously with abnormal event parameters in an AEHM as discussed in section 2.13. Finally, section 2.14 is a summary. It gives a list of desirable features in an AEHM that are stated in this chapter.

2.1 Overview of an AEHM

As mentioned, an AEHM is a control flow mechanism. The change of control flow of an execution can be divided into three steps: 1) raising an abnormal event, 2) propagating an abnormal event, 3) catching and handling an abnormal event.

An event is raised by executing a system-defined operation. This operation need not be available to programmers. Raising an abnormal event indicates the occurrence of an abnormal condition that the programmer does not want to handle via conventional flow of control. What conditions are abnormal is determined by a programmer. The execution that raises the event is the *source execution*, or simply the *source*.

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 abnormal event raised in an execution different from the faulting executing. *Propagating* an abnormal event directs the control flow of the faulting execution to a handler, and requires a *propagation mechanism* to locate a 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 abnormal events. The handler is bound to a particular set of abnormal events it handles. The faulting execution *handles* an abnormal event by executing a handler bound to the raised event. It is possible that another abnormal event is raised while executing the handler. A handler is said to have *handled* an event only if the handler returns. Unlike invoking a routine, there are multiple return mechanisms for a handler as discussed in section 2.2.1.

A *synchronous* abnormal event is one when the source and faulting execution are the same, i.e., the abnormal event is raised and handled by the same execution; an *asynchronous* abnormal event has a source different from its faulting execution. For a synchronous abnormal event, it is usually difficult to distinguish the first two steps, i.e, raising and propagating, as they are often expressed as a single operation.

Unlike a synchronous abnormal event, raising an asynchronous abnormal event does not lead to the immediate propagation of the abnormal event in the faulting execution. Using a Unix example, an asynchronous signal can be blocked, leading to a delay in propagating the event in the faulting execution. Rather, an asynchronous abnormal event is more like an asynchronous direct communication from the source to the faulting execution. The change in control flow in the faulting execution is the result of *delivering* the "abnormal event" message. In other words, the delivery initiates the propagation of the abnormal event in the faulting execution. This propagation in the faulting execution possibly can be carried out by the source or another execution but section 2.9 shows that this is very undesirable. For the moment, I assume that the source raises the event and the faulting execution propagates and handles the event.

2.2 AEHM in a language with nested blocks

John Goodenough wrote the seminal paper on exception handling[8]. In that paper, he suggested that while it is possible to bind (or attach) a handler to individual expressions and statements, it is often more desirable to bind a handler to a block to improve programmability. This design has been adopted in many languages including Ada, C++ and CLU. In addition, a handler can handle multiple events 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 abnormal event can be propagated into any block, which means that the AEHM attempts to find a handler bound to that block to handle the event. If such a handler is found, control flow is routed to that handler. If the event is propagated into an unguarded block, or a guarded block without a handler for the event, the event is propagated into the creator of the current block until a block with a handler for the event is found or all blocks are exhausted. The propagation search mechanism determines the order in which handler clauses bound to different blocks are searched.

2.2.1 Handler return mechanisms and handling models

Drew and Gough identify four abnormal event handling models in their paper[6]: the non-local transfer in `PL/I`[10], the retrying model in the language Eiffel[18], and the terminating and resumption models found in Goodenough's work[8]. An AEHM can provide more than one of the models, for example, the AEHM in Exceptional C[7].

Non-local transfer

`PL/I` 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 of a block on the stack containing the transfer point. Therefore, a label variable is not a static object. The non-local transfer in `PL/I` directs the control flow to the label variable specified in the `got` statement. A consequence of the transfer is that blocks activated after the one with the label variable are destroyed. The functions `setjmp` and `longjmp` in ANSI C together form a simplified non-local transfer facility. Calling `setjmp` is similar to setting up a label variable and calling `longjmp` performs the non-local transfer.

An AEHM can be constructed with a non-local transfer mechanism. When an abnormal event is raised, the current operation is suspended and a handler is activated. A non-local transfer in a handler can, but not always, cause the suspended operation to terminate, and hence, prevent resumption. The problem is that a non-local transfer can branch to almost anywhere making the control flow difficult to follow. This lack of discipline makes the program less maintainable and more error-prone. I do not believe this is a desirable model, and hence, it is not considered further.

Terminating model

In the terminating model, control flow continues as if the incomplete operation in the guarded block terminated without encountering the abnormal event after executing the handler. The handler, hence, can act as an alternative operation of its guarded block. The exception handling mechanism in Ada, C++ and Modula-3 adopts this model.

Retrying model

Instead of terminating the incomplete operation, the retrying model retries it. However, there must be a clear starting point of the operation in order to restart it. The beginning of the guarded block is usually taken as the starting point and there is hardly any other sensible choice. The handler is supposed to remove the abnormal condition so that the operation can complete during retry.

Gehani[7] shows that the retrying model can be implemented with an existing loop constructs found in many structural programming languages, and hence, one can say Ada, C++ and Modula-3 also provide the retrying model in addition to

```
void f1(int a[]) {
  int sum=0;
  try {
    for(int i=0,sum>100,i++)
      if (a[i]<0)
        throw Negative(i);
      else sum+=a[i];
  }
  catch (Negative(j))
    { a[j] = 0; retry; }
}
```

```
void f2(int a[]) {
  try {
    int sum=0;
    for(int i=0,sum>100,i++)
      if (a[i]<0)
        throw Negative(i);
      else sum+=a[i];
  }
  catch (Negative(j))
    { a[j] = 0; retry; }
}
```

Figure 2.1: Subtle differences between two simple retrying handlers

the terminating model. Alternatively, retrying can be seen as terminating the suspended operation followed by a jump to restart the operation.

Drew and Gough comment that the retrying model is neither widely used nor understood[6] and figure 2.1 suggests why. The syntax is an extension of C++ where the block prefixed by the keyword `try` is the guarded block with the `catch` clause representing the handler clause. Abnormal event argument binding is like ML pattern matching. There is only one handler in the handler clause in the given example. The `retry` statement terminates the handler and directs the control flow to the beginning of the guarded block. In the figure, reversing the second and third line of `f1` gives `f2` and vice versa. Obviously the two functions are different. In essence, misplacing the beginning of the guarded block is like misplacing the beginning of a loop body. However, the error involving the guarded block is more difficult to discover because the control flow involving abnormal event, and especially its propagation, is more difficult to trace. In addition, when multiple handlers exist in the handler clause using the retrying model, these handlers must use the same restart point, which may make the retrying model more difficult to use in some cases.

Eiffel is probably the first language to emphasize the use of the retrying model. However, Eiffel only allows handlers to be bound to a procedure body, which must fulfill a contract. Variable declaration in the procedure may involve some initialization and the initialization is not implicitly re-done in the case of retry. The argument for such a design is that the precondition of the contract must be satisfied before the procedure body starts execution. Satisfying the postcondition becomes the responsibility of the procedure and re-initializing should have no effects in terms of eliminating the abnormal condition — violating the contract of the procedure. Violating a precondition in a procedure call results in an exception in the caller and the procedure is not invoked. The use of contracts justifies the restriction on what can be a guarded block, and the restriction eliminates the possibility of misplacing the guarded block boundary. Furthermore, Eiffel allows only one handler for all the abnormal events in each handler clause. Consequently, the problem of multiple retrying handlers in a handler clause does not exist.

Resumption model

The resumption model continues the suspended operation after handling the abnormal event. The handler is supposed to remove the cause of the abnormal condition so that the suspended operation can continue. An alternative view is that the abnormal event represents a special request like an interrupt and the faulting execution serves the request by executing a handler. Indeed, control flow into and out of the handler is identical to that of a routine call[6, 7]. The difference is the mechanism used to locate a handler. For most imperative languages, a routine to be called is located statically, while an abnormal event handler, in general, is located dynamically. Locating a handler is discussed further in section 2.4.

2.2.2 Useful features in an AEHM

This section examines additional features that make an AEHM easier to use. The features include *abnormal event parameter*, *derived abnormal event* and *bound abnormal event*. See [2, 6, 8, 14] for in depth discussions.

Abnormal event defined as a type

An abnormal event is defined as a value of a built-in type in some languages, for example, the type `EXCEPTION` in Ada and Modula-3. Alternatively, an event is defined as a type like exceptions in C++.

The former allows defining routines for only the built-in type, but not the individual events because the signature of a routine is specified by data types. Consequently, these routines must be applicable to all declared events.

The latter is more flexible because routines can be defined specifically for an event as each event is a data type. In particular, a routine may serve as a *default handler* for a declared event. A raised event can be handled by its *default handler*[2] when the faulting execution does not have a handler for the event.

[2] I discuss handler partitioning later in this chapter. If handlers are partitioned into different types, an event can have more than one default handler as a result.

Abnormal event parameters

Abnormal event parameters enable the source to transfer important information into and out of a handler, just like routine parameters. An abnormal event parameter can be `in`, `out` or `in-out` (or equivalently read-only, write-only and read-write). While information could be passed through shared objects, abnormal event parameters reduce the amount of side effects and locking in a concurrent environment.

Synchronizing access The arguments of an asynchronous abnormal event are usually accessible to the source after the event is raised, and the faulting execution after the event is caught. Therefore, access to these arguments must be properly synchronized if references are involved. The synchronization can be a facility of an AEHM or performed by additional programming.

The former makes programming easier but can lead to unnecessary synchronization. Basically, it requires blocking the source or the faulting execution when the argument is accessed. However, there are many different kinds of synchronization like accessing shared buffer and producer-consumer. Different synchronizations have different needs and the programmer would have to provide additional information to the system.

The later is more flexible as it can accommodate specific synchronization needs. With the use of monitors, futures, conditional variables and possibly other data structures for synchronization, the synchronization required for accessing an abnormal event argument can be easily implemented as well. There is not much additional work because the other solution also requires some additional declarations to specify what synchronization is required for an argument.

I believe that the additional synchronization for accessing abnormal event arguments would be uncommon. Hence, not making it a feature of an AEHM simplifies the programming interface and hardly loses any programmability.

Derived abnormal events

Because an extensible AEHM allows definition of new abnormal events, an *abnormal event hierarchy* is useful to provide better name space management, similar to a class hierarchy found in object-oriented languages. An abnormal event can be derived from another abnormal event (its parent event) just like deriving a subclass from its parent class. An event is an *ancestor* of another event if 1) the former is the parent of the latter, or 2) the former is an ancestor of the latter's parent. An event is a descendent of another event if and only if the latter is an ancestor of the former. A descendent is more specific than any of its ancestors. A programmer can now choose to handle an event at different degrees of specificity. Hence, *derived abnormal events* support a more flexible programming style.

A design issue about derived abnormal events is whether to allow derivation from multiple (parent) events. The issue is similar to choosing single or multiple inheritance. Cargill[4] and a few others[22] argue against multiple inheritance as a general programming facility. My focus, however, is on derived abnormal events.

Consider the example of deriving an abnormal event `network-file-err` from `network-err` and `file-err`[14]. Superficially, this may look reasonable but there are subtle problems. Let there be a handler clause with a handler for `network-err` closing the network connection, and a handler for `file-err` that closes the file. Note the handler clause might not have a handler for `network-file-err` because the handler clause was written before introducing the event `network-file-err`. 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? Executing both handlers may look legitimate, but indeed it is not. If a handler clause only has a handler 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 event having multiple parent events as one of its parent event may not be appropriate. If an event cannot be handled as its parent event, the derivation becomes imperceptible.

I strongly believe that *multiple derivation* — deriving an event from multiple events — is not a good feature for derived events in an AEHM as it introduces significant complications into the semantics.

Bound abnormal events and conditional handling

It is sometimes useful to have a handler handling events raised related to a particular object. For example, a handler only wnats to catch a `file-err` event for file `file1` but not of any other files because the guard block to which the handler binds created `file1`. In this case, binding the handler to an abnormal event bound to an object prevents the handler from catching the same abnormal event but bound to a different object. The handler is said to handle a *bound abnormal event*.

It is possible to derive a new event `file1-file-err` from `file-err` and write a handler for the new event. However, this practice implies a new event for every object, including dynamically created objects, and can explode the total number of abnormal events.

```
{
  SomeObject a = new SomeObject();
  SomeObject b = new SomeObject();

  try {
    // guarded block
  }
  catch(DerivedEvent &d) when (d.obj == b) {
    // handling bound DerivedEvent if obj is b
  }
  catch(ParentEvent &p) when (p.obj == a) {
    // handling bound ParentEvent if obj is a
  }
}
```

Figure 2.2: An example of bound event

Bound abnormal events can be mimicked by passing the associated object as an argument to the handler and checking if the argument is the right object before handling it; if the argument is not the right object, the event is re-raised[2]. Note the re-raising is merely a coding convention. In general, requiring programmers to follow coding conventions is very unreliable.

In addition, the mimicking becomes infeasible with the use of derived events. Suppose that in figure 2.2, `DerivedEvent` derives from `ParentEvent`. `ParentEvent` has a field `obj` for its bound object. The boolean expression after the optional keyword `when` specifies the object to which the event must be bound for the handler to catch the event. If `DerivedEvent` bound to object `a` is raised, the second handler is chosen. If both `when` clauses are missing, or equivalently, without bound abnormal events, the first handler is chosen. After catching the event, the first handler discovers that `(d.obj!=b)`, and consequently re-raises the event. The second handler cannot handle the re-raised event simply because the first handler is not part of the guarded block. Therefore, this "catch first, then re-raise" approach is not a substitute for bound abnormal events.

Extending this useful idea of bound abnormal event, it is reasonable to specify an arbitrary condition for a handler to be selected. The idea is similar to guarded entry in Ada and μC++ for conditional synchronization — I call it *conditional handling*. Bound abnormal event is a special case of conditional handling. Unless explicitly stated, I do not separate conditional handling from bound abnormal events in this thesis.

2.2.3 Questionable features in an AEHM

Some features in certain AEHM's are questionable in their usefulness, namely, the *single level (propagation) mechanism* in CLU[15], the abnormal event *specification list* in Modula-3[3], and a *handler for multiple events*.

Single level mechanism

In the *single level mechanism*, an abnormal event cannot propagate beyond the invoker of a procedure. The fundamental argument for the single level mechanism is that the invoker must know the complete behaviour of a procedure — the mapping which the procedure represents. If an abnormal event cannot be handled by the invoker, it is converted into a failure event, which removes any chance of handling the specific event at a higher level.

Specification list

A *specification list*[3] is a part of the procedure's signature. It specifies what abnormal events the procedure may propagate to its invoker. It is considered a runtime error if a procedure tries to propagate an abnormal event that is not in its specification list, and the consequence is probably aborting the faulting execution or the whole environment.

Essentially, a specification list allows an abnormal event to propagates through many levels of procedure invocation only as long as it is explicitly stated that propagation is allowed.

Drawbacks of single level mechanism and specification list Both the single level mechanism and the specification list are trying to precisely specify the behaviour of a routine. While specification of routine behaviour is certainly essential, I believe that these two features are too restrictive, especially with programming facilities designed for code reuse. They can only be used in an embedded environment where extensibility is less of an issue.

Consider the C++ template function `sort`

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

using the operator function `<`. It is practically impossible to know what abnormal events may be propagated from the function `<`, and subsequently those from `sort` because `sort` is supposed to take many different `<` functions to support code reuse. Therefore, it is impossible to give a specification list to the template function.

[3]The specification list is equivalent to the throw list in C++ in a routine's signature. A major difference, though, is that a throw list is optional while a specification list is not.

An alternative is to force the full specification only at instantiation of the template function, but not with the generic procedure itself. However, if a language allows arguments of function pointers and/or polymorphic methods or routines, the various instantiations may be difficult to check at compile time. Hence, this is not a legitimate solution. For similar reasons, the single level mechanism does not suit a polymorphic programming style either.

Yet another alternative is to treat the specification list as part of the function signature, just like the parameter list, but this eliminates many possibilities for code reuse like the template function `sort`. While type checking the parameter list is reasonable because arguments are used in every call, abnormal events are rarely used and the reduction in code reuse by typechecking the specification list is not justified.

Finally, specifying the events that can be propagated from a routine becomes more difficult with the introduction of asynchronous abnormal events because it is impossible to pre-determine when an asynchronous abnormal event may be propagated.

Therefore, I do not think these two features are beneficial to the design of an AEHM in a general programming environment.

Handler for multiple events

Ada allows a handler to handle more than one event in the absence of derived abnormal events. With derived abnormal events, this feature becomes much less useful, if not useless. Note that a handler handling one event and its derived event is thought to handle one event only because a derived event is an instance of its parent event. It is difficult to imagine two unrelated abnormal events being handled by a single handler. Furthermore, a handler for multiple events is not compatible with abnormal event parameters, which Ada does not support.

2.3 Stack unwinding

On detecting an abnormal condition while executing a block, it is sometimes necessary to terminate an incomplete operation in the faulting execution when continuing this operation is unsafe. Indeed, one of the requirements of an AEHM is the ability to terminate this suspended, incomplete operation. In an imperative programming language, terminating an operation is accomplished by destroying one or more blocks associated with the incomplete operation. This destruction of stack frames is commonly known as *stack unwinding*.

2.3.1 Stack unwinding in propagation

Stack unwinding is often done as part of the propagation process, so there are two different ways of propagating an abnormal event: *throwing* an abnormal event is propagating an abnormal event and unwinding the stack; *signalling* an abnormal event is propagating without stack unwinding.

Normally, the propagation mechanism begins by throwing or signalling and the semantics continue until an appropriate handler is found. However, it is possible to override signalling by throwing and vice versa using simple handlers as shown with pseudocode in figure 2.3. Overriding signalling by throwing starts the stack unwinding process by first destroying the stack frame of the handler(which is just

```
// override signalling
try {
  guarded region that
  signals an event
}
catch (any_event e) {
  throw the event e
}
```

```
// override throwing
try {
  guarded region that
  throws an event
}
catch (any_event e) {
  signal the event e
}
```

Figure 2.3: Handlers overriding default propagation mechanism

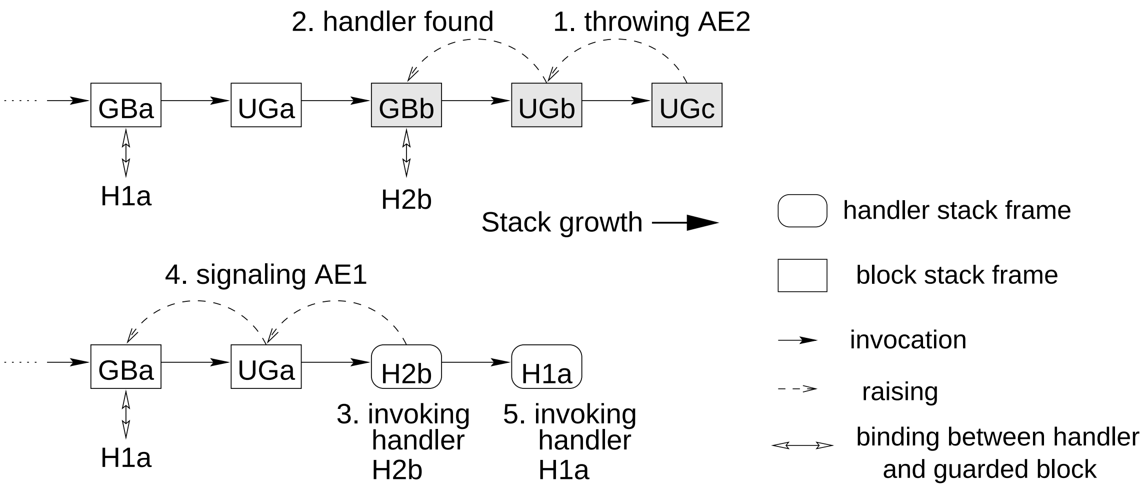


Figure 2.4: Effect of overriding throwing by signalling

a stack frame on the top of stack), then the stack frame of the block that originally signalled the abnormal event. Overriding throwing by signalling cannot recover the destroyed stack frames as a result of prior stack unwinding. An example is illustrated in figure 2.4. In step 1, abnormal event AE2 is thrown in UGc. Step 2 is the propagation which finds handler H2b in GBb. In step 3, the handler H2b overrides the throwing by signalling and stops the stack unwinding. Notice the shaded stack frames have already been destroyed and cannot be recovered. The signalling of AE1 in step 4 eventually invokes the handler H1a of GBa. Once H1a terminates, regardless of its handling model, control flow returns to H2b. Also note that when H2b terminates, regardless of its handling model, flow control returns to the point in UGa as if invocation of GBb terminates normally as long as H2b does not throw an event. Handling a thrown event with the resumption model and handling a signalled event with the terminating model are discussed in section 2.7.

In general, the need to override is rare and the conditions making overriding necessary are specific to the local context. Therefore, a programmer is responsible for specifying when the propagation mechanism should change using handlers like those in figure 2.3.

Choosing a propagation mechanism

Given the choice between throwing and signalling, an AEHM must let the programmer specify the appropriate propagation mechanism. Two approaches for binding a propagation mechanism to an event are discussed here: binding when declaring the event and binding when raising the event.

Associating the propagation mechanism with an abnormal event during its declaration forces a partitioning of abnormal events, as in μSystem[2] with exceptions and interventions, and Exceptional C[7] with exceptions and signals. An abnormal event can either be thrown or signalled depending on its type but not both. An overloaded `raise` statement is sufficient because an event determines the subsequent propagation mechanism.

Alternatively, the binding can be delayed until the event is raised. Hence, an event can be thrown or signalled at different times. Since the event itself does not determine the propagation mechanism, a `throw` and a `signal` statement are necessary. It should be emphasized that in this case, the mechanism is chosen when raising the event, not when propagating the event. Although the idea was introduced by Goodenough[8], for some reasons, I have yet to find an AEHM with abnormal events that can be both thrown and signalled.

2.3.2 No stack unwinding in propagation

Another possibility is to further delay stack unwinding until a handler is being executed. Hence no stack unwinding is done during propagation. Consequently, stack unwinding has to be done within the handler.

One design option is to provide an `unwind` statement to trigger the stack unwinding as in VMS[11]. The `unwind` statement might take an argument which specifies the number of stack frames to be unwound. Obviously, this `unwind` facility is difficult to use because a programmer has to determine how many blocks to be destroyed at runtime. The other option is to provide a `terminate` statement to trigger stack unwinding after handling the event, and a `resume`[4] statement for the resumption model. The `terminate` statement does not require an argument specifying how many stack frames to unwind; the AEHM determines that. Essentially, the handler makes the decision whether to unwind a stack with the choice of `resume` or `terminate` and there is only signalling of an event. The absence of throwing prevents the source from unwinding the stack and consequently opens the door for unsafe resumption.

[4]The `resume` statement functionally is identical to a `return` statement in a routine but different from the `resume` statement in a coroutine.

Table 2.1: Summary of the propagation options with regards to stack unwinding

Option	forms of raise statement	unsafe resumption
Specify event to be raised or signalled when defining events	one overloaded raise statement for throwing and signalling	impossible
Specify event to be raised or signalled when raising a event	two different raise statements for throwing and signalling	impossible
Initiate stack unwinding in a handler by a `terminate` or `unwind` statement	one statement for signalling, no throwing mechanism	possible

Table 2.1 summarizes the various design options in an AEHM with regard to stack unwinding.

2.4 Existing propagation models

The propagation mechanism determines how to find a handler. Most of the AEHM's adopt *dynamic propagation*, which goes up the invocation hierarchy to find a handler. The other propagation mechanism is *static propagation*, which goes up the lexical hierarchy. Static propagation was proposed by Knudsen[12, 13], and his work has been ignored in the literature on AEHM, including[2, 6, 7, 20]. As a result, dynamic propagation is often known as propagation and I have used propagation to refer to dynamic propagation so far. The language BETA[17] has an AEHM based on Knudsen's idea.

2.4.1 Dynamic propagation

Dynamic propagation allows the handler clause bound to the most recent block in the invocation hierarchy to handle the abnormal event, provided it has an appropriate handler that can handle the event. A consequence is that the event is handled by a handler closest to the block where propagation of the event starts. Usually, an operation higher in the invocation hierarchy is more general while those lower in the hierarchy are more specific. Handling an abnormal event at the lowest level deals with the abnormal event in a context that is more specific, without affecting the abstract operation at a higher level. The handling of an abnormal event 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. First, there is the *visibility* problem. Second, the *dynamic choice* of a handler goes against the principle of a statically scoped programming language. Third, dynamic propagation can cause *recursive signalling.* These criticisms are discussed one by one before looking at a *static exception handling mechanism* with *static propagation*, a proposal intended to solve the problems of dynamic propagation.

Visibility Dynamic propagation propagates an unhandled abnormal event to its invoker, and since an invoker can be in a different lexical scope, an abnormal event can be propagated into a scope where the event is invisible — as the event is not declared in the lexical scope of the invoker — and then back into a scope where it is visible. It has been mentioned that this property of losing and regaining identity is undesirable because a routine can now propagate an abnormal event that it does

```
try {
  throw IOerror();
}
catch (IOerror &ioerr) {
  ...
}
```

Figure 2.5: Static choice of handler

not know at all[20]. Some language designers believe that once an abnormal event is propagated into a scope where it is invisible, it should lose its identity forever. This is equivalent to propagating a failure event found in the single level mechanism.

During the discussion on specification list and single level mechanism in section 2.2.3, I clearly state how propagating an event dynamically to a block in which the event is unexpected is beneficial to code reuse, and the drawbacks of prohibiting it. This visibility argument against dynamic propagation is essentially saying that dynamic propagation is bad because an invisible event is not expected to be raised. Following the conclusion in section 2.2.3, the visibility criticism is invalid.

Dynamic choice of handler The handler chosen to handle an abnormal event with dynamic propagation often cannot be determined statically. A rare example where the handler of an event can be determined statically is shown in figure 2.5. With dynamic propagation, a programmer seldom knows statically what action is actually taken to handle an event when the event is raised. Hence, the program becomes more difficult to trace and the AEHM becomes harder to use, as some people have stated[17, 12, 20].

Properly using an AEHM requires only raising an abnormal event on the occurrence of an abnormal condition. Raising an abnormal event should not be able to specify the action to be taken. Otherwise, it is unnecessary to define the handler in a separate place — bound to a guarded block higher in the invocation hierarchy. Therefore, the dynamic choice of a handler or the uncertainty of handling action when an event is raised is not introduced by a specific AEHM but by the nature of the problem and its solution. For example, a library may declare abnormal events and raise them without providing any handlers so that the library clients can provide a specific handler to handle the event. The library programmer cannot know how the event is handled after it is raised. Interestingly, the inability to trace the execution beyond encountering an abnormal condition when writing a reusable library is not limited to AEHM's only. Returning an error value in the status return value technique does not allow the library writer to know the action by the library client on receiving the error value. What becomes necessary is to establish proper abstraction for various program elements. Indeed, properly using an AEHM can provide better communication for abnormal conditions between a library and its clients than the status return value and status flag techniques.

In general, an AEHM should not be hard to use and understand. The difficulties in understanding the control flow of a program using an AEHM are not a direct result of using AEHM but rather a result of the nature of the problem — providing an alternative to communicate abnormal conditions between a caller and its callee. When using the AEHM facilities correctly, the control flow of a program should be understandable, especially since abnormal events occur infrequently.

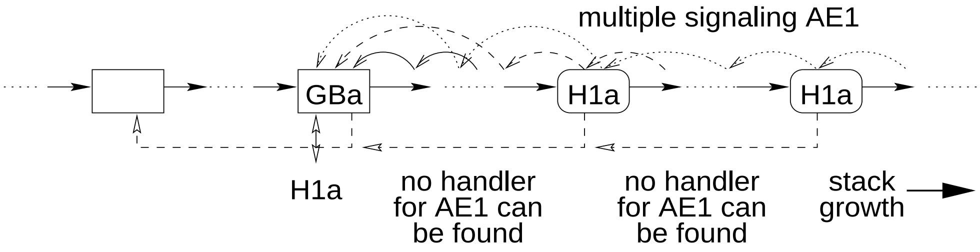


Figure 2.6: An example of recursive signal

Recursive signals MacLaren briefly discussed the recursive signalling problem in the context of the language `PL/I`[16], but the problem is also found in Exceptional C and μSystem. All these languages ignored the issue of recursive signalling. The language Mesa[19] made a relatively unsuccessful attempt to solve this problem because its solution is often criticized as incomprehensible, unuseful and not justified for its cost[22]. This following only discusses the problem of recursive signalling. The Mesa solution and some other possible solutions are discussed in section 2.11.

Signalling does not unwind the stack. In figure 2.4 on page 27 an event AE1 is signalled and handler H1a is invoked. Suppose that AE1 is signalled by some routine again before H1a completes, and assume that the execution flow does not set up a new handler for AE1 after the first signalling of AE1, as in figure 2.6. As a result, the propagation mechanism chooses H1a again and this can continue until the runtime stack overflows. This situation is *recursive signalling.* Recursive signalling is similar to infinite recursion, and is difficult to discover both at compile time and runtime because of the dynamic choice of a handler.

While an event may be raised synchronously or asynchronously, I only consider synchronously signalled events as candidates for causing recursive signalling. A synchronous event is related to the faulting execution context. In other words, a synchronous event is related to other actions performed by an execution. Since an asynchronous event is not raised by the faulting execution, it is difficult to associate the event with the faulting execution. Furthermore, for asynchronous events to cause a stack overflow, they must be generated at a rate faster than can be handled by the faulting execution. However, this stack overflow can be avoided if the faulting execution could run faster. Hence this problem is a result of limited computing resources rather than dynamic propagation.

Recursive signalling is probably the only legitimate criticism against dynamic propagation. I propose a solution to recursive signalling in section 2.11.3.

2.4.2 Static propagation

Knudsen proposed a *static exception handling mechanism*[12] as an alternative approach with the intention of resolving the dynamic choice of handler problem. He used *sequels*[23] as exception handlers. A *sequel* resembles a procedure in many ways. It has parameters and a *sequel call* transfers control to the sequel just like a procedure call. However, when a sequel terminates, execution continues at the end of the block in which the sequel is declared. Thus, handling an abnormal event with a sequel adheres to the terminating model. Within a nested block language with static scoping, calling a sequel `IOerror` invokes the sequel with that name in the closest lexical block. If `IOerror` is a procedure instead of a sequel, handling the event with `IOerror` adheres to the resumption model. The propagation of the abnormal event is along the lexical hierarchy, i.e., using *static propagation*, because of name binding resolution in statically scoped languages. In the static exception handling mechanism, for each raise, the handling action is known at compile time. The implementation of static propagation makes use of the normal lexical links among blocks available at runtime in any statically scoped programming language.

During the development of the language BETA[17], whose exception handling mechanism is based on Knudsen's idea, it became obvious that the original static exception handling mechanism was insufficient. I believe the insufficiency is caused directly by the static propagation mechanism. Subsequently, a new concept *virtual sequel* was introduced[13]. The action of a virtual sequel depends heavily on the call hierarchy.

Not only is the static scheme insufficient, it also has two additional drawbacks. First, *an abnormal event cannot be propagated to a lexical sibling*, even though an abnormal event visible to a lexical parent is also visible in its siblings. Second, *an abnormal event cannot be propagated outside a lexical hierarchy*. These drawbacks are elaborated in the followings with figure 2.7 as an example.

Deficiencies of static propagation In figure 2.7, a name in the name hierarchy is composed of a sequence of letters. Each name begins with its parent's name, and a letter is appended to distinguish among siblings. In addition, let e be an abnormal event visible to `A`, `B` and `C` and all their descendents. Furthermore, assume that there is no handler for e bound to any of the names. Finally, assume the sequence of invocations on the right of the figure. The dashed lines denote the lexical links needed for accessing non-local references in the nested static scopes.

Suppose `ABC`, `AB` and `A` terminate without raising an abnormal event in the figure. If `BAC` raises e, it propagates to the stack frame of `BA` but not to `BAB` because the

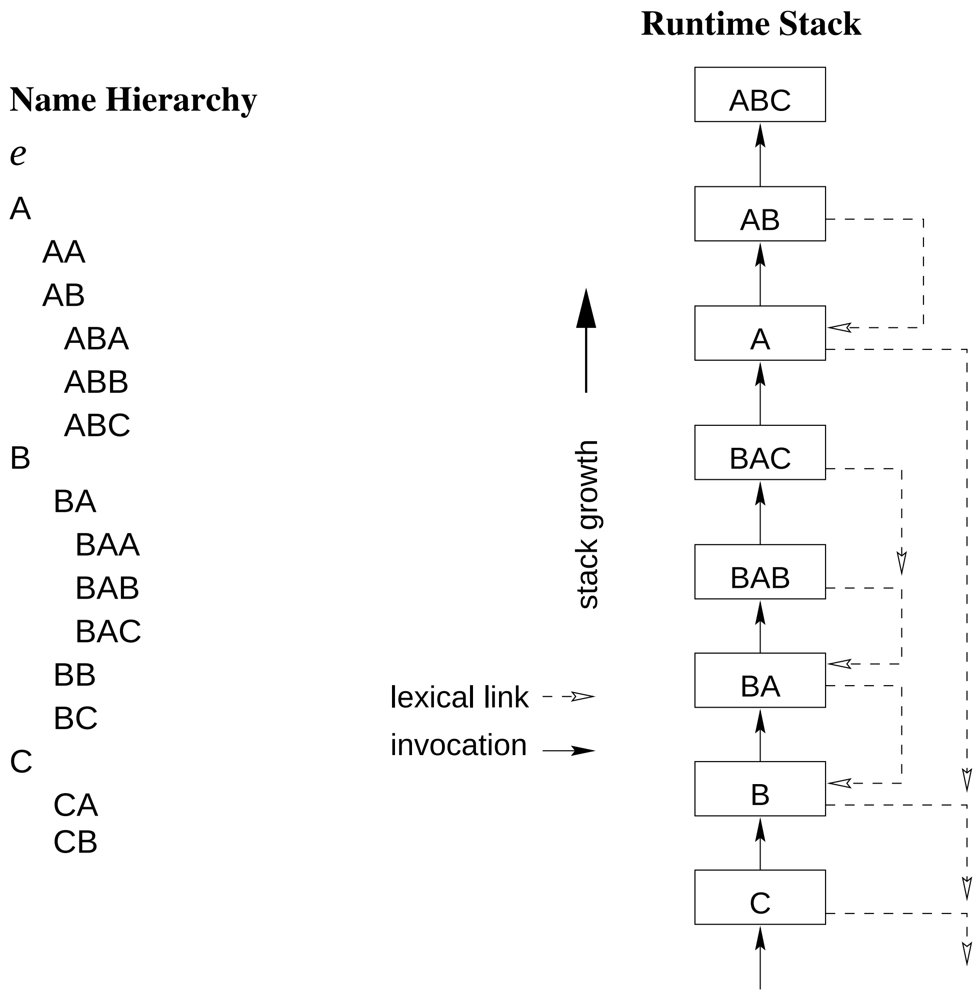


Figure 2.7: Drawbacks of static propagation

propagation follows the lexical link. This is the first drawback.

Alternatively, if the stack frame of `ABC` raises e instead of terminating normally, the event is propagated upwards along the lexical hierarchy, i.e., from `ABC` to `AB` to `A`. The other lexical hierarchies not containing `ABC` can never handle e, even though e is visible, which is the second drawback.

Notice that static propagation never suffers recursive signalling because an event signalled inside a handler always invokes a handler **strictly**[5] closer to the lexical root than the current one.

The introduction of virtual sequel in BETA shows the insufficiency of a general static abnormal event handling mechanism, and more importantly, the necessity of the dynamic choice of a handler. Therefore, a handler will very rarely be coded as in figure 2.5.

2.5 Context of a handler

This section examines how stack unwinding affects the context of a handler.

2.5.1 With stack unwinding

Although both C++ and Ada adopt the terminating model, the context of their handlers differ. A handler in C++ executes in a scope outside its guarded block, while in Ada the guarded block is nested inside, and hence, can access variables declared in it. Nonetheless, this does not result in any functional difference with regard to unwinding.

[5]The word "strictly" describes that an event signalled in a handler cannot be handled by a handler in the same lexical level.

```
BEGIN
  VAR x:INTEGER;
  -- invisible in handler
  BEGIN
    VAR x:INTEGER;
    -- visible in handler
       .
       .
       .
  EXCEPTION
    WHEN Others =>
       x := 0;
  END;
END;
```

```
{
  int x;
  // invisible in handler
  { int x;
    // visible in handler
    try {
       .
       .
       .
    }
    catch (...)
      { x = 0; }
  }
}
```

Figure 2.8: Handlers in Ada and C++ : identical semantics

Figure 2.8 is a simple example illustrating that by moving the handler and possibly adding another nested block, the same semantics can be accomplished in either language. This strategy works as long as a handler can be associated with any nested block.

According to [14], the approach in C++ can lead to better use of registers. Therefore, local declarations in a guarded block are assumed to be invisible throughout the rest of the discussion, unless specified otherwise. Though the previous example uses the terminating model, the technique of moving declarations up and down in the lexical hierarchy applies to the resumption model and the retrying model as well.

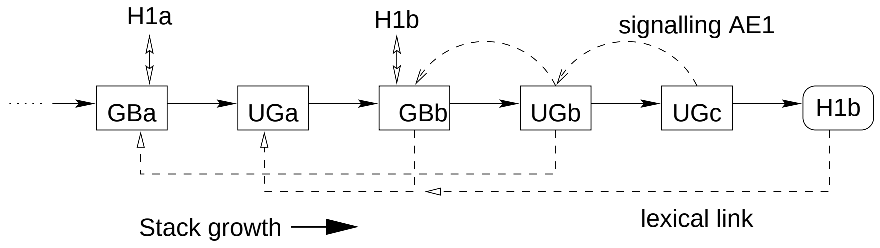


Figure 2.9: Using resumption model to handle a signalled event

2.5.2 Without stack unwinding

A handler should be statically scoped, just like other blocks in a language with nested blocks. To be consistent with the terminating model, the handler is not nested inside the guarded block, and hence, the variables declared in the guarded block are invisible to the handler. However, the variables are not destroyed because there is no stack unwinding as shown in figure 2.9 when UGc signals AE1. If the bindings in the guarded block GBb are visible in the handler H1b as in Ada, the lexical pointer from H1b would point to GBb instead of UGa.

Figure 2.10 contains a C++ like program with a nested function declaration and the portion of the runtime stack after an invocation of `f(true)`. The handler in the figure is assumed to adopt the resumption model. Looking at the runtime stack illustration, the value of `x` printed should be `true`. However, in ordinary recursive routine invocation, including indirect recursion, a variable or parameter name declared in an instance of routine `f` is hidden immediately after another invocation of `f` until that invocation terminates. Using this criteria, the binding of `x` to `true` is hidden after the invocation of `f(false)`. Nonetheless, the former binding to `true` is visible in the handler. The handler in figure 2.10 is able to bypass the most recent invocation of `f`. The bypass is a result of using a context

```
void f(bool x) {
  void nested() {
    try {
      f(!x);
    }
    catch (E) {
      cout << x << endl;
      // is x true of false?
    }}

  if (x)
    nested();
  else
    signal E();
}
```

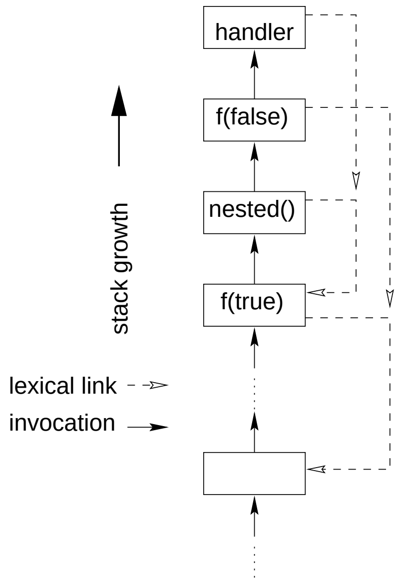


Figure 2.10: Static scoping in a resuming handler

different from what the runtime stack reflects.

Section 2.2.1 states that invoking a handler in the resumption model is identical to calling a routine. This statement should be interpreted carefully, taking the subtle difference between the context of a statically chosen routine and a dynamically chosen handler into account.

The difference in context can happen in the terminating model and the retrying model as well, if an AEHM does not unwind the stack while propagating the event. The bottom line is the absence of throwing even when invoking a terminating or retrying handler. As the stack is not unwound, the runtime stack structure, when invoking the terminating or retrying handler, is identical to invoking a resuming handler until the system unwinds the stack after executing the handler. In addition, recursive signalling is possible when using a terminating or retrying handler

```
terminate(AE ae) {
  cleanup(ae.x);
}

resume(AE ae) {
  recover(ae.x);
}
```

(a) partitioned

```
handler(AE ae) {
  if (resumable) {
    recover(ae.x);
    resume;
  }
  cleanup(ae.x);
  terminate;
}
```

(b) not partitioned

Figure 2.11: A handler that can terminate and resume

if unwinding is absent. Signalling an abnormal event while a terminating handler has been invoked can possibly invoke the same handler again.

A programmer should rarely be concern about the difference in scoping because the handler is still statically scoped. An exception is walking up and down the stack during debugging. One must understand the context of a resuming handler.

2.6 Design options in handlers

An AEHM might force a handler to bind to one and only one handling model at compile time. In other words, a handler would be declared as either a *terminating handler* or a *resuming handler* but never both. See the left example in figure 2.11. Hence the handlers would be partitioned according to the handling model. The AEHM's in Exceptional C[7] and μSystem[2] have this *handler partitioning.*

With handler partitioning, a handling model must have been 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. The handler can decide how to handle the event depending on the information passed to the handler or by checking the global context, as shown in the right example of figure 2.11. Note that additional statements are necessary to specify when to resume and when to terminate. The VMS and the BETA exception handling systems are two examples of this approach. Obviously, more flexible handlers can be written without handler partitioning.

2.7 Matching and unmatching handling

Two propagation mechanisms, throwing and signalling, are discussed in section 2.3. Since throwing implies terminating the current operation, it is sensible to handle it with the terminating model. However, other possibilities for handling an event in the presence of throwing and signalling must be examined. For example, a handler may decide to resume after catching a thrown event.

Section 2.2.1 suggests that there are essentially two handling models — the terminating model and the resumption model — because the retrying model is a special form of the terminating model.

The terminating model and throwing both imply terminating an operation. The resumption model and signalling imply the operation is resumed. Therefore, it is sensible to handle a thrown event with the terminating model, and a signalled event with the resumption model. I call these two cases *matching handling*. *Unmatching handling* is counter-intuitive but can be valid. The following examines unmatching handling of thrown and signalled events.

2.7.1 Unmatching handling of thrown event

When an event is thrown, the stack is immediately unwound and the operation cannot be resumed. Handling a thrown event with the resumption model, therefore, cannot resume the "terminated" operation. It does not matter whether the handler is declared as resuming, or tries to resume at runtime.

However, if the operation throwing an event expects a handler to provide an alternative for itself, and the resuming handler catching the event expects the operation raising the event to continue, the result is misleading and difficult to understand, possibly resulting an error which may only be noticed long after the handler returns. I believe that unmatching handling of thrown event is an unsafe feature.

2.7.2 Unmatching handling of signalled event

A handler intending to terminate has four possibilities after handling a signalled event:

1. The stack is not unwound and the event is handled with the resumption model, i.e., the termination is ignored.

2. The stack is unwound only after the handler executes to completion.

3. The stack is unwound by executing a special statement during execution of the handler.

4. The stack is unwound after finding the terminating handler but before executing it.

The first option is the only one that does not terminate after executing the terminating handler. This option favours the *source* choosing resumption while the others favour the *faulting execution* choosing termination. I believe that termination should be chosen over resumption for unmatching handling of a signalled event for safety's sake. Terminating a resumable operation at most wastes computing power; resuming a non-resumable operation is unsafe. Though signalling implies that the suspended operation is resumable, if the terminating handler does not carry out necessary actions to make the resumption safe, the first option opens the possibility of unsafe resumption. Therefore, it is dismissed.

The next two options can result in recursive signalling in the terminating handlers. The problems can be avoided by the fourth option which unwinds the stack before executing the handler. Indeed, a programmer does not realize when stack unwinding takes place, either while an event propagates or after finding a handler; the last option essentially handles the signalled event as a thrown event. The last option also simplifies the task of writing a terminating handler because a programmer does not have to be concerned if the stack is unwound inside a terminating handler. This is significant because stack unwinding has side effects.

As it appears to be superior to the other two options favouring termination, the last option is chosen to be the semantics of unmatching handling of signalled events.

2.7.3 Abstraction for handling models

Table 2.2 summarizes the semantics of all matching and unmatching handling of abnormal events. In matching handling, one can determine what model (and the

Table 2.2: Matching and unmatching handling

	handler using terminating model	handler using resumption model
signalled event	handled with termination model like a throw event (unmatching handling)	handled with resumption model (matching handling)
thrown event	handled with termination model (matching handling)	handled with termination model (unmatching handling)

control flow) is used to handle a raised event by knowing either how an the event is raised or what handler is chosen. Abstracting the resumption model and the terminating model are done in a symmetric fashion.

The same cannot be said about unmatching handling. In particular, one cannot tell whether a signalled event is handled with the resumption model without knowing what handler catches it, but a thrown event is always handled with the termination model. Throwing and signalling are asymmetric in unmatching handling. Without knowing the handling model used for a signalled event, it becomes more difficult to understand the signalling mechanism than the throwing mechanism and the signalling mechanism in matching handling. I believe that unmatching handling is inferior to matching handling.

2.8 Selecting a handler

The propagation mechanism determines how handler clauses are searched to locate a handler. It does not say which handler in a handler clause should be chosen to handle an event, if there are multiple handlers capable of catching the event.

This section first discusses issues about three orthogonal criteria — specificity of a handler, agreement and closeness — for choosing a handler among those capable of handling a raised event, followed by issues on missing an appropriate handler. A handler can handle an abnormal event if it is bound to the event or to one of its ancestors. In addition, if the handler is bound to an object as described in section 2.2.2 on page 21, the object to which the raised event is bound must match to the one specified by the handler.

2.8.1 Three orthogonal criteria

A handler clause can have several handlers capable of handling one event, because of the existence of derived events and bound abnormal events. A handler clause can have a handler for an event and another for its derived event; it can also have a handler for a bound abnormal event and another one that is not bound. An eligible handler is more *specific* than another (in any handler clause) if:

1. they both handle the same event and the former applies conditional handling while the other does not (e.g, the former handles a bound abnormal event and the latter does not), or
2. the former handles an event derived from the one handled by the latter, and both these handlers handle the event *conditionally* or *unconditionally*[6].

Specificity is desirable in handling an abnormal event. It is sometimes infeasible to tell which handler in a handler clause is more specific. In particular, a handler for

[6]The differentiation can be further refined by taking the conditional expressions into account, i.e., to determine if one implies the other. The handler with the implying expression should be more specific than that with the implied expression. However, my definition is sufficient to show that conditional handling can affect specificity of a handler.

an abnormal event AE and a handler conditionally handling an ancestor of AE is equally specific to handle an event AE of the object.

With throwing and signalling as the propagation mechanisms, and handler partitions, matching handling can be considered as an *agreement* to terminate or resume an operation between the source and the faulting execution upon handling an abnormal event. This agreement is also between the code raising the event and the code handling the event. Choosing a handler that matches the the propagation mechanism maintains this *agreement* and is desirable for similar reasons to the idea of programming with contract in Eiffel[18] and routine specification. Unmatching handling does not provide this agreement. Agreement is only applicable in an AEHM with the two propagation mechanisms and the handler partitioning.

The *closeness* factor says that the closest handler capable of handling an abnormal event is chosen. A handler is closer than another if its handler clause is searched prior to others based on a given propagation mechanism.

A language designer has to set priorities among these three orthogonal criteria and there are six ($_3P_3$) possible ways to order them. In addition, the priority of handling a thrown event is orthogonal to that of a signalled event. Consequently, there are actually 36 (the square of six) possible ways to define how to select a handler. Instead of looking at individual selecting schemes, I examine the priorities first.

2.8.2 Setting priorities for the criteria

Specificity is good, but putting it before *closeness* complicates the semantics of derived events and the implementation. A handler can only protect an event from

```
try {
  try {
    throw AE2; // AE2 is derived from AE1
  }
  catch(AE1) {
    // code for handler
  }
}
catch(AE2) {
  // code for handler
}
```

Figure 2.12: Example showing the order of priority

propagating out of a guarded block if there is no more specific matching handler further away. Figure 2.12 illustrates why this semantics is undesirable. The event AE_2 is derived from AE_1. If specificity has a higher priority than closeness, the handler for AE_2 is chosen, not the one for AE_1. As event derivation cannot be anticipated in general, no handler can guarantee to handle all the events and prevent them from propagating further and affecting more abstract operation. Indeed, a library routine working perfectly if invoked by one client routine can behave differently if invoked by a different client routine because the client routine sets up its handlers differently. I believe this semantics is a trap for programmers and potentially breaks down any abstraction. In addition, before committing to a particular handler, the runtime system has to search through other handler clauses to ensure a more specific handler is not available. Though an AEHM putting specificity before closeness can definitely be implemented, the searching mechanism is obviously harder and more costly to implement and the additional resources that it requires seem unjustified. Choosing a more specific handler over a closer handler for a thrown event results in more wasted computation, and potentially irrecoverable data, as the stack is unwound further.

Agreement should have a higher priority than *closeness* (i.e., the highest priority) when applicable because unmatching handling is less comprehensible, and hence, less programmable because of the poor abstraction discussed in section 2.7.3. Indeed, *agreement* should be **mandatory** for thrown events because of safety concern discussed at the end section 2.7.1. Also, if agreement is not mandatory, then throwing becomes more complicated to implement taking into account the scenario in which the stack should not be unwound beyond the closest resuming handler if a terminating handler is not found[7]. It appears that the pros of this *mandatory agreement* outweighs its cons. *Mandatory agreement* is also extended to signalled events for consistency. Section 2.8.3 shows that this extension does not result in a loss of functionality, namely, using a terminating handler to handle a signalled event when necessary.

An AEHM with mandatory agreement can be thought to have a terminating handler hierarchy and a resuming handler hierarchy. A *handler hierarchy* is built on the *closeness* of the handlers. Throwing searches a handler in the former and signalling in the latter.

Because two handlers in the same handler clause can be equally specific to a raised event, the system must provide 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 list of the handler clause. Whatever these

[7]The scenario can also happen when agreement does not have the highest priority among the three criteria, or equivalently, when unmatching handling of thrown event is allowed.

additional criteria are, they should only be applied to resolve ambiguity after using the other three ordering criteria: agreement, closeness and specificity. Furthermore, the programmer should be able to control the resolution, like placing the handlers in a specific order in a handler clause, because the additional criteria used are only conventions.

2.8.3 Using a default handler

I mention in section 2.2.2 that routines can be defined for an event and an event-specific routine can be chosen as a default handler if an execution does not have a handler for a propagated event. With mandatory agreement, it is reasonable for an event to have a default terminating handler and a default resuming handler.

A default terminating handler allows to clean up the environment on an uncaught thrown event. Consider using the default terminating handler of an event declared in a library to clean up internal objects in the library. The advantage of this practice over relying on an execution to catch the event and execute a routine is better abstraction and less user intervention.

If a signalled event is not handled in an execution, the default resuming handler for the event is executed. A question is whether the execution should be resumed after executing the default "resuming" handler. However, as default handlers are associated with an event, the question needs no specific answer. The default handler can abort the execution, throw an event (possibly the same event) to terminate the resumable operation but not necessarily the execution, or simply resume by ignoring the event.

Figure 2.13 shows that default handlers can be defined in C++ using virtual

```
class ParentEvent {
public:
  virtual void defaultResumeHandler();
  // defaultHandler aborts the program
}

class ChildEvent : public ParentEvent {
public:
  virtual void defaultResumeHandler();
  // defaultHandler invokes a debugger
}
```

Figure 2.13: Defining default handlers in C++

method. Details about default handlers as well as the event declarations are omitted. The virtual method `defaultResumeHandler` is invoked automatically when no handler can catch a raised event. By overriding the virtual method, a default handler can be defined specifically for every event.

Executing a default resuming handler that merely throws the same event triggers another search in the terminating handler hierarchy. The effect is handling the signalled event with a terminating handler. I want to emphasize that this is only a programmable option. It does not violate the *mandatory agreement* because of the explicit throw in the handler.

2.9 Concurrency and handling abnormal events

With multiple executions and possibly multiple threads, different executions can carry out different operations associated with handling one single abnormal event. For example, an execution raising an abnormal event, known as the source in section 2.1, can ask another execution to propagate an abnormal event in a third execution, the faulting execution, which eventually handles the abnormal event.

2.9.1 Concurrent environment

Race conditions are a fundamental issue in a concurrent environment. It happens when at least two different executions try to modify a shared resource. The solution is to ensure mutual exclusion via locking. An execution must lock a shared resource before modifying it. If the resource is already locked, the execution must block and wait.

Consider the situation where an execution is able to directly propagate an abnormal event in another execution — the faulting execution — in a concurrent environment. By definition, propagating an abnormal event requires changing the execution state — including the runtime stack and the program counter — of the faulting execution. Consequently, the run-time stack and the program counter become shared resources. Threads are no longer independent entities as executions in one thread can interfere with another thread. To avoid corrupting the execution state, locking is inevitable. Hence, an execution would have to lock its own run-time stack before calling a procedure. Obviously, this approach is extremely inefficient with the large number of lockings to deal with a situation that occurs rarely. Therefore, it is reasonable that only the faulting execution is allowed to propagate the abnormal event in a concurrent environment.

2.9.2 Coroutine environment

In a coroutine environment, locking the execution state before modifying it is unnecessary. Race conditions do not exist as there can only be one running execution.

```
try {
  // guarded region GRa
  try {
    // guarded region GRb
  }
  catch (EventB eb) { ... }
}
catch (EventA ea) { ... }
```

Figure 2.14: Possible undesirable behaviour in coroutine environment

However, even without the possibility of race conditions, it is still a bad idea to allow an execution other than the faulting one to propagate an abnormal event. Consider the program fragment in figure 2.14. Suppose execution EX_1 is suspended in the guarded region `GRb`. While it is suspended, another execution EX_2 propagates abnormal event `ea` in EX_1, which directs the flow control of EX_1 to the handler of guarded region `GRa`(unwinding the stack in the process). Before EX_1 starts running again, a third execution EX_3[8] propagates abnormal event `eb`. Hence, the flow control of EX_1 goes to another handler determined in the dynamic context(further unwinding the stack). The net effect is that neither of the abnormal events is handled by any handler in the program fragment.

The alternative approach is for EX_1 to propagate the abnormal events. Regardless of which order EX_1 raises the two arriving events, at least a handler for one of the events is invoked.

Because of these confusing situations, I believe that only the faulting execution should be able to propagate an abnormal event, as for the concurrent environment. Since a concurrent environment can be composed of many coroutines, this common

[8] EX_2 and EX_3 do not have to be distinct.

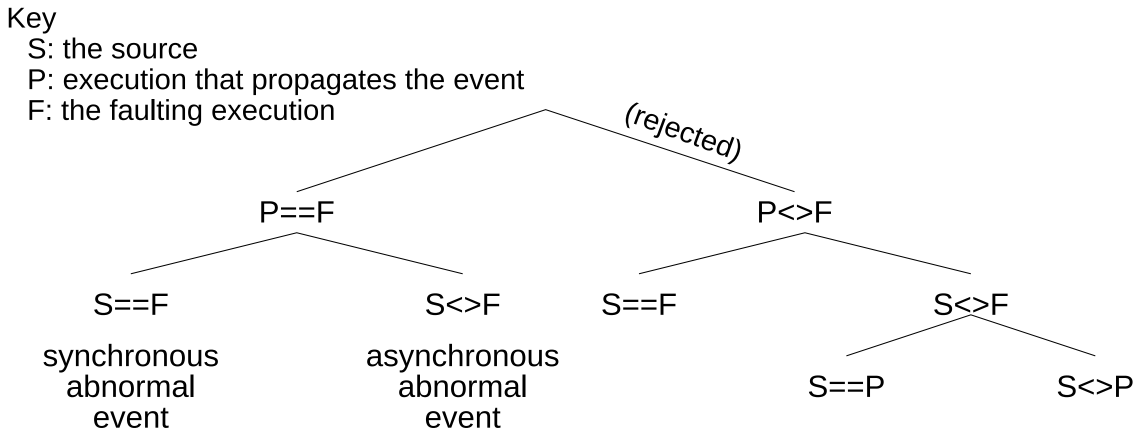


Figure 2.15: Abnormal events in an environment with multiple executions

restriction gives overall consistency.

2.10 Issues specific to asynchronous events

As only the faulting execution can propagate an abnormal event, there are only two possibilities left, as shown in figure 2.15, how an event is raised, propagated and handled. *Synchronous events* are abnormal events raised by the faulting execution. *Asynchronous abnormal events* are abnormal events raised by a different execution — the source and the faulting execution are different. A common example of asynchronous abnormal event is Unix signals.

The remainder of this section first looks at the communication protocol for implementing asynchronous abnormal events. The problems of using asynchronous abnormal event are then discussed. Then special attention is given to hardware interrupts.

2.10.1 Communication requirement

An asynchronous abnormal event has its source different from its faulting execution. Because only the faulting execution can propagate the abnormal event and directly alter the control flow, the source must inform the faulting execution to propagate the event. This consequence is a form of direct communication since the communication has a clear receiver and does not involve any shared object. A message is transmitted from the sender (in this case the source) to the receiver (the faulting execution).

There are two major categories of direct communication: synchronous and asynchronous communication. *Synchronous communication* requires the sender block until the receiver is ready to receive the message. *Asynchronous communication* does not block the sender. In both cases, the receiver is blocked if the sender has yet to send a message.

Requirement imposed by the source

Using synchronous communication implies that the source is blocked until the faulting execution executes a complementary receive call. However, an execution may infrequently (or never) check for incoming abnormal events. (Receiving an abnormal event message is discussed next.) Therefore, the source can be blocked for an extended period of time waiting for the faulting execution to receive the abnormal event message. Therefore, synchronous communication is dismissed. Asynchronous communication has a non-blocking send, and hence, allows a the source to raise an abnormal event at multiple executions without suffering the extended delay.

Requirement imposed by the faulting execution

Nonetheless, the asynchronous communication for abnormal events must be different from ordinary asynchronous communication. In the latter case, a message is delivered only after the receiver executes some form of `receive` command. The former requires the receiver to receive a message without explicitly executing a `receive` because an AEHM should preclude checking for an abnormal condition, in this case the infrequent arrival of an abnormal event message. The programmer is only required to set up a handler to handle a rare condition. From the programmer's perspective, the delivery of an asynchronous abnormal event (message) is transparent. Therefore, the underlying system must poll for the arrival of new messages, and propagate the corresponding abnormal event on the arrival of the message. The delivery of abnormal events must be timely, although not immediate.

There are two polling strategies: *explicit polling* and *implicit polling*. Explicit polling requires the programmer to insert explicit code to activate the polling, such as executing a non-blocking `receive` or a blocking `receive` with timeout. Implicit polling is the opposite; polling is performed by the underlying system. Hardware interrupts involve implicit polling because the processor of the machine automatically polls for the arrival of its interrupts.

Explicit polling gives the programmer control over when an asynchronous abnormal event can be raised. Therefore, the programmer can delay, or even completely ignore the handling of asynchronous abnormal events. Delaying and ignoring asynchronous abnormal events are both undesirable. The other drawback of explicit polling is that the programmer has to worry about when to and when not to poll.

Implicit polling alleviates programmers from polling, and hence, provides an apparently easier interface to programmers. The programmer does not worry about when to poll for asynchronous abnormal events. On the other hand, implicit polling has its own drawbacks. First, infrequent implicit polling can still delay the handling of asynchronous abnormal events; polling too frequently can deteriorate the runtime efficiency. Without specific knowledge about a program, it is difficult to have the right frequency of implicit polling. Second, implicit polling suffers the *reentrant problem*, which is discussed in section 2.10.2.

Therefore, I believe an AEHM with asynchronous abnormal events should employ both implicit and explicit polling. Implicit polling reduces the degree of damage that a programmer can do to the execution environment by ignoring asynchronous abnormal events. 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, there are certain situations where implicit polling is very undesirable. They are mostly low-level applications where execution efficiency is very important and the programmer is supposed to have extensive knowledge of the programming task. Therefore, it is definitely beneficial to be able to turn off implicit polling, probably by a compiler or runtime switch, to address the needs of these specific applications.

2.10.2 Reentrant problem

An imperative program often relies on side effects and state information to carry out its computation. Portions of the program must be carried out atomically for correctness. An example is the `malloc` function in `C`. An implementation of `malloc` works correctly in a pure sequential environment where there is no asynchronous abnormal events. However, the implementation may not work when the execution of the program can be arbitrary interrupted by an asynchronous abnormal event. Suppose the execution is updating some state information for memory management and it is interrupted by an asynchronous abnormal event. The memory management system can be in an inconsistent state. If the handler for the abnormal event invokes `malloc` — a reentrant invocation of malloc, the inconsistency in the memory system can corrupt the computation. The lack of guarantee of uninterruptable execution causes the *reentrant problem*. A function like `malloc` is said to be *non-reentrant* when its correctness relies on its uninterrupted execution and yet this uninterrupted execution is not guaranteed by its implementation. Any implicit polling may interrupt a non-reentrant operation.

The reentrant problem should not be mistaken as a race condition. Arbitrary interruption by asynchronous abnormal events is the cause of the reentrant problem, while multiple threads competing for shared resources results in a race condition. Preventing another execution from accessing a shared resource by locking can avoid the race condition, but does not guarantee reentrance. To ensure the correctness of a non-reentrant routine, the execution must block the delivery, and consequently the propagation, of asynchronous abnormal events, hence temporarily precluding interrupts.

I have considered hardware interrupts as implicitly polled asynchronous abnormal events so far. The implicit polling is done by the CPU. However, these interrupts can interrupt a language operation like a routine invocation that is supposed to be atomic. Its effects on a programming language are similar to asynchronous abnormal events.

The reentrant problem solution relies on blocking interruptions. As (hardware) interrupts can happen even at times when asynchronous abnormal events cannot, it is more difficult to ensure proper atomicity with interrupts. The following sections discuss programming with asynchronous abnormal events followed by interrupts.

2.10.3 Blocking asynchronous abnormal events

While blocking asynchronous abnormal events is essential, an execution should be able to selectively block some abnormal events but not all. If execution can block an event `AE`, there are two different semantics with derived abnormal events: 1) the execution blocks only `AE`, 2) the execution blocks `AE` and all its descendants[9].

Blocking an individual event but not its descendents, known as *individual blocking*, can make programming tedious as the programmer must list all the events being blocked. Not only that, individual blocking does not complement the abnormal event hierarchy. If a new derived event should be treated as an instance of its ancestors, the event must be blocked wherever its ancestors is blocked. Individual blocking does not automatically block the descendents of the specified events, and therefore, introducing a new derived event probably requires modifying existing code in order to prevent the new derived event from activating a handler bound to its ancestor when the handler should not be allowed to execute.

The other alternative, *hierarchical blocking*, blocks an event and all its descendents. The derivation becomes more restrictive because a derived event also inherits

[9]The two are equivalent in a scheme without derived events. An example is the UNIX signals.

the "blocking" characteristics of its parent. Compared to individual blocking, hierarchical blocking is more complex to implement and probably has a higher runtime cost as well. However, the improvement in programmability makes hierarchical blocking attractive.

There is another criteria for selective blocking. A priority can be assigned to each event and a derived event can override its parent event's priority when necessary. Selective blocking can be achieved by blocking out abnormal events of priority lower than or equal to a specified value. This selective blocking scheme trades off the programmability and extensibility of the hierarchical blocking for lower implementation and runtime cost.

The problem with priorities is that two different events can have the same priority. Introducing a new abnormal event now requires an understanding of its abnormal nature plus its priority compared to other events. Hence, defining a new event requires an extensive knowledge of the whole system, which makes the system less maintainable and understandable. It is also possible to add priorities to hierarchical selective blocking; a programmer needs to specify the event and the priority in order to block an asynchronous abnormal event. However, it does not resolve the problem of maintaining consistent priorities throughout the abnormal event hierarchy. In general, abnormal event priority is an additional concept that increases the complexity of the overall system.

I believe that hierarchical blocking with derived abnormal events is better than the other approaches in an extensible abnormal event handling mechanism. Note that multiple derivation in section 2.2.2 complicates hierarchical blocking, and the same arguments can be used against hierarchical blocking with multiple derivation.

Turning blocking on and off

A language can provide explicit routines to turn on and off the blocking of an asynchronous abnormal event. The programming style is very similar to that of using semaphore for locking and unlocking, and is not a very good abstraction. Programming errors resulting from forgetting a complementary call are difficult to debug.

An alternative is to provide a new block structure called a *protected block*, which specifies a list of asynchronous event to be blocked. When the execution enters a protected block, its blocking of asynchronous events is modified, which is reset when it exits the block. The effect is like entering a guarded block.

Another alternative is to turn on the blocking effect by a special routine but turn it off automatically on exiting the block where the routine call is found. Compared with protected blocks, using a single routine call gives a simpler structure to a block as it avoids introducing the concept of protected blocks and reduces the number of nested blocks. However, a routine call is syntactically less distinctive, which I consider a disadvantage.

I believe that providing a protected block should be the simplest and most consistent in an imperative language with nested blocks. Only the first approach — explicit turning on and off blocking — is obviously inferior to the others. Note that, regardless of how to turn it on and off, blocking should be off initially for all but a few special abnormal events to ensure that an execution has a chance to set up handlers.

2.10.4 Multiple pending asynchronous abnormal events

No matter how often polling is done, there is a chance that multiple asynchronous abnormal events arrive at the same execution between two polls. An asynchronous abnormal event waiting to be raised by the faulting execution is described as *pending*.

Since every asynchronous abnormal event is sent via an asynchronous message, the order of arrival can be arbitrary due to delays and parallelism among executions. The order of arrival can be chosen to determine the order of handling pending events, which gives a sequency in event delivery. However, a strictly FIFO message delivery order is unacceptable as illustrated by the following example. An execution can receive a first event and blocks its delivery. Due to the FIFO delivery, events arriving later cannot be delivered before this event and must remain pending. Consequently, the delivery of an asynchronously thrown event to prevent the execution from continuing an erroneous computation can be delayed for an extended period of time.

It is possible to eliminate pending asynchronous abnormal events by not queueing abnormal event messages. In other words, each execution has a buffer for only one pending abnormal event. The buffer is overwritten after it is filled, or overwritten only if the new message has a higher priority, or new messages are discarded after the buffer fills. The risk of losing an asynchronous abnormal event can make a system less robust. Hence queuing abnormal event messages is superior.

Yet an AEHM can provide a more flexible semantics for handling pending abnormal events using a user-defined priority scheme. Section 2.10.3 discusses how a priority scheme reduces extensibility. However, I do not believe it is an appropriate solution in an environment emphasizing code reuse.

Proposed delivery order for multiple pending events

It appears that FIFO order based on event arrivals should be acceptable for its simplicity in understanding and low implementation cost. Nonetheless, allowing a pending event whose delivery is blocked to prevent delivering other pending events seems undesirable at times. Hence, an event should be able to be delivered before earlier events if the eariler events are blocked.

This out of order delivery has important implications on the programing model of asynchronous abnormal events. First, the programmer must be aware of the fact that two abnormal events having the same source and faulting execution may be delivered out of order (when the first is blocked but not the second), which means the second event cannot be considered as a consequence of the first. This approach may seem a bit unreasonable, especially when causal ordering is proved to be beneficial in distributed programming. However, out of order delivery is acceptable for emergency messages and abnormal events can represent emergency situations. For example, an asynchronous abnormal event may be used to terminate the current computation of the faulting execution and this event is better delivered as soon as possible.

Considering the rarity of abnormal events, I believe that FIFO delivery order should be followed in general. Out of order delivery is only allowed when events arrived earlier are blocked. Nonetheless, the most adequate delivery scheme remains as an open subject, and the answer may only come with experience.

2.10.5 Reentrancy with interrupts

When a hardware interrupt occurs, an interrupt service routine (ISR) is invoked immediately if the interrupt is not blocked[10]. If it is blocked, the result can be an immediate loss of the interrupt, a pending interrupt, which can be overwritten by another interrupt, or a pending interrupt in a interrupt (priority) queue. However, the exact semantics of interrupt blocking does not affect reentrancy.

Some interrupts require a immediate service and their ISR has no side effect on the faulting execution. The interrupt can be thought as "robbing" a thread, any thread, to execute a service routine. The execution of these ISR's is transparent to the faulting execution and does not cause a reentrant problem.

Other interrupts aim to modify an execution either via side effect or changing control flow. It is possible to direct an interrupt to a particular execution but the exact mechanism of directing an interrupt to an execution has no effect on reentrancy. The execution state is modified as a result of executing the ISR. Programming with interrupts is difficult because interrupts can happen while the runtime stack and/or the heap is in an inconsistent state.

One way to ensure reentrancy is to use blocking facilities provided by the system. However, this solution is definitely unacceptable. First, blocking an interrupt can possibly lose the interrupt. In the worst case, an execution must turn on blocking before initiating the construction or destruction of a stack frame and turn it off immediately after modifying the stack, resulting an enormous runtime overhead. The blocking of interrupts can be very long during stack unwinding.

[10]Blocking an interrupt is more commonly known as masking an interrupt, but the former is used throughout this thesis to emphasize that interrupts and abnormal events are related.

Converting interrupts to abnormal events

Another solution is to convert interrupts into language level asynchronous abnormal events. The implementation is outlined here. When receiving an interrupt, an interrupt routine is invoked immediately. The ISR sends an asynchronous abnormal event message to the faulting execution directly, or via an intermediate execution. In the latter case, the intermediate execution later forwards the message to the intended execution. Regardless of whether an intermediate execution is used, interrupts must be blocked when enqueuing and dequeuing the message to avoid the possibility of corrupting the message queue by another interrupt or the execution processing the asynchronous events. The delivery of the message at the faulting execution can now be controlled by facilities in the AEHM, and hence, reentrant problems caused by interrupts can be avoided.

Converting interrupts to abnormal events still requires blocking interrupts at times when the abnormal event message queue is being manipulated and possibly results in losing interrupts in some systems, but these systems would already lose interrupts.

The conversion also simplifies the interface within the language. The interrupts and blocking of interrupts can be completely hidden within the AEHM and programmers only need to handle abnormal conditions at the language level. This approach also improves portability across operating systems.

Also note that only those interrupts modifying an execution state require blocking. Hence, the language runtime kernel can hide unnecessary interrupts from programmers, although the language should allow a user to have some degree of control over hiding certain interrupts. The visibility of interrupts can be modified for in-

```
void ISR_stub() {
  // ISR_stub is the general structure of an ISR
  try {
    // start of ISR body
    ...
  }
  terminate(AnyThrowEvent) {
    // AnyThrowEvent is the root of all thrown event
  }
  resuming(AnySignalEvent) {
    // AnySignalEvent is the root of all signal
    // events the following event can be caught by
    // the above terminating handler
    throw AnyThrowEvent;
  }
}
```

Figure 2.16: Preventing thrown events from escaping an interrupt service routine

dividual execution. An invisible interrupt is also described as transparent as the interrupted execution is not aware of the invocation of the ISR.

This scheme also allows an execution to raise an interrupt to another execution instead of an abnormal event via the operating system as the interrupt can be converted to an abnormal event. However, sending interrupts does appear to be inferior, as interrupts do not usually have arguments to pass additional information.

Interrupt service routines and abnormal events Activating an ISR for a transparent interrupt can result in raising an abnormal event. If the ISR is invoked on the interrupted execution's stack, this abnormal event can be propagated back to the interrupted execution. Figure 2.16 shows how to prevent propagating the event to the interrupted execution using the existing AEHM features. Indeed, Ada 95 specifies that propagating an exception (thrown event in Ada 95) from an interrupt handler has no effect. The rationale is that an interrupt handler is invoked by an imaginary execution rather than the interrupted one, and therefore, an interrupted execution should not be affected by interrupts[9]. Within the Ada 95 context, all interrupts are always asynchronous and transparent. Furthermore, they are distinct from exceptions. The no-effect semantics seems acceptable. In addition, Ada 95 does not consider interrupts as a possible communication media among executions.

Nonetheless, I think that abstracting interrupt services is worth further investigation and the Ada 95 always-transparent semantics may be too restrictive[11]. The question that has to be answered is whether propagating an abnormal event, regardless of being thrown or signalled, from an ISR can benefit programming. Allowing such propagation implies that interrupts are mostly transparent to the interrupted execution but in some rare cases, the ISR does communicate with the interrupted execution via abnormal events.

Apparently, in Ada 95, an ISR can only communicate with the interrupted execution via shared objects, which possibly provides a way to emulate the effect of propagating an abnormal event from the ISR to the interrupted execution. On the other hand, emulating in the reverse direction seems easier as shown in figure 2.16, and hence, seems to be more favourable. Though I prefer the latter to the former, the possibility of giving up too much safety cannot be ignored.

---

[11] I must admit that my understanding of Ada 95 design in system and real-time programming is incomplete. Hence my comment on Ada 95 may be unfair.

2.11 Preventing Recursive Signalling

Dynamic propagation can cause recursive signalling as described in section 2.6. The AEHM in Mesa[19] is probably the only AEHM that attempted to solve this problem. The rest of this section looks at the solution in Mesa, and other possible solutions.

2.11.1 Hybrid propagation

Though the static exception handling mechanism in section 2.4.2 on page 37 has serious drawbacks, its static propagation model eliminates any chance of recursive signalling of synchronous abnormal events. It may appear that by properly combining the use of dynamic propagation and static propagation, a new *hybrid* propagation mechanism can eliminate the drawbacks of the individual propagation mechanisms.

In general, dynamic propagation is good except for the possibility of recursive signalling. However, not all abnormal events signalled by a resuming handler cause recursive signalling. Even if a resuming handler re-signals the event it handles, which guarantees activating the same resuming handler again, (infinite) recursive signalling may not happen because the handler can take a different execution path as a result of modified execution state.

Therefore, it is difficult to determine when to use static propagation and dynamic propagation without additional knowledge about the program, and sometimes the knowledge is not available when writing a reusable library. Providing two propagation mechanisms merely complicates the AEHM interface to programmers. I believe that mixing the propagation mechanisms does not provide an easily applicable solution to recursive signalling.

2.11.2 The Mesa propagation

The Mesa[19] semantics for what should happen when a signal is raised synchronously, while handling another, is not very clear according to Alan Freier[12] and is difficult to understand. However, I believe Mesa uses a *simple marking strategy* to solve the recursive signalling problem.

A simple Mesa program in figure 2.17 with output[13] illustrates the marking of handlers in Mesa. The keyword `PROC` declares a procedure object. The main program is bound by `BEGIN` and `END`. An `ENABLE` clause can appear at the beginning of a block; it serves as the handler clause in Mesa. The `RESUME` and the `CONTINUE` statement can only appear in a handler. The former specifies resumption while the latter termination.

Whenever the procedure `Test` is invoked, a new instance of its handler clause is created as well. Once a handler in a handler clause is invoked, it is marked and cannot be invoked again until the mark is clear when this invocation terminates. Here is the outline of the control flow of the program. After invoking `Test[1]` and `Test[2]`, `Sig1` is raised. The handler for `Sig1` in `Test[2]` is invoked, printing out the first line. The handlers in the same handler clause for `Sig2` and `Sig3` are similarly invoked. When handling `Sig3`, `Sig1` is raised again, which invokes the

[12]Most of the information on Mesa I collected is from email communications with individuals who have extensive experience with the language. Alan Freier points out that nested signals in Mesa, which he considers as a "(mis)feature", is poorly documented.

[13]Code and output is provided by Michael_Plass.PARC@xerox.com via email.

```
-------------  subprogram  -------------

Sig1: SIGNAL = CODE;
Sig2: SIGNAL = CODE;
Sig3: SIGNAL = CODE;
RaiseSig1: PROC = { SIGNAL Sig1 };
RaiseSig2: PROC = { SIGNAL Sig2 };
RaiseSig3: PROC = { SIGNAL Sig3 };

Test: PROC [nest: INT] = {
  ENABLE {
    Sig1 => { PrintLn["Sig1 at A"]; RaiseSig2[]; RESUME };
    Sig2 => { PrintLn["Sig2 at A"]; RaiseSig3[]; RESUME };
    Sig3 => { PrintLn["Sig3 at A"]; RaiseSig1[]; RESUME };
    };
  IF nest < 2 THEN Test[nest+1] ELSE RaiseSig1[];
  };

BEGIN
  ENABLE {
    Sig1 => { PrintLn["Sig1 at top"]; CONTINUE };
    Sig2 => { PrintLn["Sig2 at top"]; CONTINUE };
    Sig3 => { PrintLn["Sig3 at top"]; CONTINUE };
    };
  Test[1];
  END;


-------------  output of subprogram  -------------

Sig1 at A
Sig2 at A
Sig3 at A
Sig1 at A
Sig2 at A
Sig3 at A
Sig1 at top
```

Figure 2.17: Handler marking in Mesa

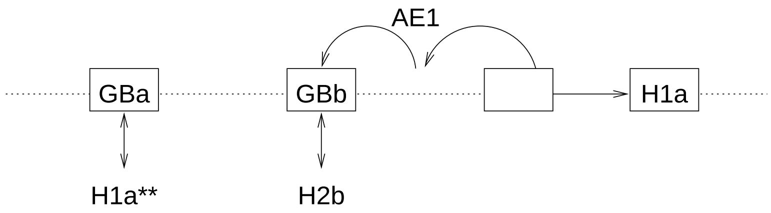
handler for `Sig1` in `Test[1]` because the one in `Test[2]` is still marked, causing the repeated output as shown. `Sig1` is then raised again, handled by the handler in the main program. The continue statement subsequently terminates the program.

This scheme prevents recursive signalling by not reusing a handler clause bound to a specific invoked block. The propagation mechanism always starts from the top of the stack to find an unmarked handler for a signalled event. However, this unambiguous semantics is often described as confusing and incomprehensible.

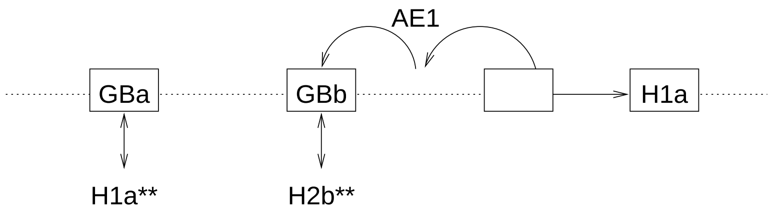
Look at the example in figure 2.18(a)(a), which shows the runtime stack of a Mesa execution. Each dotted line represents a sequence of omitted stack frames and only stack frames of interests are named. GBa and GBb are guarded blocks with a set of resuming handlers. In the figure, handler H1a (handler for AE1 bound to GBa) is invoked and marked (with "**") because of the signalled event AE1. If AE2 is raised before H1a returns, H2b is chosen to handle the event.

For Mesa, signalling an event in a guarded block and in one of its handlers can invoke different handlers, even though the guarded block and its handlers are in the same lexical scope. For instance, in figure 2.18(a), an event from H1a can be handled by handlers bound to the callees of GBa, but one from GBa is handled by H1a or handlers bound to the callers of GBa. Clearly, the lexical scoping does not reflect the difference in semantics.

Not only that, the semantics is undesirable. Procedural abstraction states that GBa should be treated as a "client" of routines that it invokes directly or indirectly. This client should have no knowledge about the implementation of what it uses. However, if signalling from H1a (or any other resuming handler) is a useful feature, some knowledge about the handlers bound to a stack frame between GBa and H1a



(a) Mesa



(b) μC++

Figure 2.18: Marking handlers

must be available when writing the handler as well as its guarded block because signalling an event in H1a may invoke a handler between GBa and H1a. Therefore, the Mesa signalling mechanism conflicts with procedural abstraction.

Moreover, abnormal events are designed for communicating abnormal conditions from callee to its caller without the drawbacks of return values. What Mesa does, as illustrated by the example, for a re-signalled event inside a resuming handler is more abnormal condition propagating from caller to callee than vice versa.

2.11.3 Improving the Mesa propagation

The propagation mechanism discussed in section 2.4 searches for a handler by simply going up the runtime stack one stack frame at a time. This simple mechanism has the recursive signalling problem. Mesa eliminates the recursive signalling problem by not reusing a handler bound to the same guarded block. This solution, however, has complicated semantics.

I propose a new propagation mechanism that solves the recursive signalling problem. Similar to that of Mesa, the new propagation mechanism also employs marking to avoid reusing a handler. Further, the mechanism is extended to cover asynchronous events, which Mesa does not have. Before looking at the mechanism in detail, the concept of *consequent events* is defined, which helps to understand why the semantics of the new propagation mechanism is desirable.

Consequent events

An execution raising an abnormal event synchronously indicates that an abnormal condition is encountered. A handler can catch an event and then raise another synchronous event when encountering another abnormal condition, resulting in a second synchronous abnormal event.

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 abnormal event being handled in the execution (if there is one). A consequent event of AE is either the immediate consequent event of AE, or the immediate consequent event of another consequent event of AE. The consequence relation is transitive, but not reflexive. The only events that are not a consequent event are asynchronous events, and synchronous events that are propagated when no other events are being handled. An asynchronous abnormal event is not considered as a consequent event of other abnormal events propagated in the target execution because the condition resulting in the event is encountered by another execution, and in general, is not related to the target execution. Any synchronous event raised before a handler activated by the last asynchronous event terminates is a consequent event of the asynchronous event.

The new propagation mechanism

In the new propagation mechanism, agreement is still mandatory if applicable; closeness is more favourable than specificity. Hence, the propagation mechanism can only serach one handler hierarchy even when there are two.

The propagation mechanism goes up the chosen handler hierarchy one level at a time as it normally does in a dynamic propagation scheme to find a handler capable of handling the event being propagated. In addition, all the handlers in the level being "visited" are marked[14] for the event and its consequent events, regardless if a handler is found. The mark is clear only if the event is handled, meaning that the handler that caught the event returns, i.e., terminates or resumes.

So, how does this propagation mechanism make a difference? Figure 2.18(b) on page 73 shows the runtime stack of the same example in figure 2.18(a) with the new propagation mechanism. The only difference in the figure is that handler H2b is also marked when AE1 is propagated. Hence, even if H1a synchronously signals AE2, the event cannot be handled by H2b, or any other resuming handlers bound to a stack frame between GBa and H1a, inclusive. When handling an event, the flow of the execution can enter additional guarded blocks. When the execution encounters another synchronous (signalled or thrown) abnormal event, the handlers of these guarded blocks are first examined as they are not marked. If no appropriate

[14]The meaning of marking is same as that in Mesa.

handler can be found, the event is propagated to the original handler, and then to the invoker of its guarded block.

The modification does not affect the throwing mechanism because once a set of handlers are marked, the handler hierarchy shrinks because of stack unwinding. Hence, the propagation mechanism can be consistently applied to both throwing and signalling.

The mechanism eliminates recursive signalling as a resuming handler marked for a particular event cannot be invoked to handle its consequent events. Not only that, propagating a synchronously raised event out of a handler does not invoke a handler bound to a stack frame between the handler and its guarded block, similar to an event propagated out of a guarded block. In other words,

With this propagation mechanism, a resuming handler can be treated as a callback routine bound to a guarded block, which gets invoked by a signalled event. A handler (synchronously) signalling a event is trying to invoke a callback provided by the clients (or callers) of its guarded block.

If a resuming handler throws an event, a terminating handler bound to a callee of the guarded block of the resuming handler can be invoked. I believe that this is different from the case of signalled event because the flow of control does not return to the resuming handler after the thrown event is handled[15]. After all, it does not hurt to get a chance to clean up before terminating the callee.

No handler can be marked for an asynchronous event being propagated because an asynchronous event is not a consequent event. Therefore, the propagation mech-

[15]The case with thrown event is like "I cannot do what you ask me to"; signalling is like "do something for me first before I try to do what you ask".

anism finds a handler by going up every guarded block in the runtime stack. Hence, a handler not eligible to handle an event and its consequent events can be chosen to handle a recently arrived asynchronous event, reflecting the lack of consequentiality of asynchronous events.

The propagation mechanism is clearly better than other existing propagation mechanisms because:

- it supports throwing and signalling and the search for a handler is uniformly defined for both,
- it prevents recursive signalling and handles synchronous and asynchronous abnormal events according to a sensible consequence relation among abnormal events,
- the context of a handler closely resembles its guarded block as reflected by their lexical location; in particular, an event propagated out of a handler is handled as if the event is directly propagated out of its guarded block.
- it is compatible with the well understood throwing mechanism, and extends the semantics to the signalling mechanism – a clear indication that nested events should be more comprehensible than nested signals in Mesa.

2.12 Abnormal event name space

This section discusses two issues about the abnormal event name space. The first is partitioning of the abnormal events into throw-only, signal-only and dual events; the second is on an abnormal event hierarchy.

2.12.1 Partitioning of abnormal events

During the discussion of propagation mechanisms in section 2.3.1, an abnormal event can be tied to a particular propagation mechanism at declaration. A consequence is the partitioning of abnormal events. This section examines this abnormal event partitioning.

An abnormal event can be declared to be *throw-only* or *signal-only*, if it can only be thrown or signalled respectively, or *dual* if it can be both thrown and signalled. Without partitioning, every event becomes a dual event if the AEHM supports throwing and signalling.

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

Having dual events in addition to throw-only and signal-only events enhances the AEHM programmability.

First, encountering an abnormal condition can lead to signalling an abnormal event or throwing one depending on the execution context. Without dual events, two different abnormal events must be declared, one being signal-only and the other throw-only. These two events are apparently unrelated without an additional naming convention. Using a single dual event is simpler.

Next, using a dual event instead of a signal-only event for some abnormal conditions allows the event to be thrown if the event is signalled and no resuming handler is set up in the execution to handle the event. This can be done by defining a default resuming handler for the event which throws the event. A signal-only event cannot be thrown.

Finally, always restricting one raising mechanism to one abnormal event has its drawbacks. Suppose a throw-only abnormal event is declared in a library. A client of the library has to throw the event even if the client wants to signal the event in its context. Signalling a different event may not allow easy communication with other clients of the library. The problem is that throw-only and signal-only events lack the flexibility of dual events, and flexibility improves reusability.

2.12.2 Abnormal event hierarchy

With derived abnormal events, a language designer must decide whether one kind of event can be derived from another, say a signal-only event from a throw-signal event. This is *heterogeneous* derivation, as opposed to *homogeneous* derivation, where derived events must be the same type as the parent event.

Homogeneous derivation is simpler to understand; on the other hand, heterogeneous derivation is more flexible because it allows deriving an event from any other kind of event. It is possible to put all the abnormal events in one hierarchy with heterogeneous derivation.

Without any restriction on heterogeneous derivation, a throw-only event can be derived from a signal-only event and vice versa. With handler partitioning, a terminating handler still cannot catch a signalled event as unmatching handling of abnormal events is rejected. Heterogeneous derivation does not affect the flow control of an AEHM and the only notable advantage of such heterogeneous derivation is in hierarchical blocking, making it easier to block events of the same kind regardless how one may be raised. Yet I do not think that this is a big advantage.

Other forms of heterogeneous derivation involve dual events. A heterogeneous derivation with a parent dual event is troublesome. The child event is either throw-only or signal-only and can be caught by a handler for the parent dual event. The handler can throw or signal the dual event but the child event can only either be thrown or signalled. Throwing a signal-only event or signalling a throw-only event in the handler is definitely undesirable. On the other hand, there is neither an obvious problem nor advantage if the dual event is the child event rather than the parent event.

It seems that heterogeneous derivation does not simplify programming and may confuse programmers. Hence I do not believe that it is a useful feature.

2.13 Event parameters in derived events

Abnormal event parameters are useful in passing information between different parts of a program, i.e., where an event is raised and handled. They help limit the need for shared objects. However, with an abnormal event hierarchy, they cannot be simply treated as data fields in a class in a class hierarchy. First, an example is given to illustrate the problem.

A handler for AE can be considered as a mapping with the following signature:

$$\mathrm{AE_{in}} \rightarrow \mathrm{AE_{out}}$$

where $\mathrm{AE_{in}}$ and $\mathrm{AE_{out}}$ are information passed into and out of the handler respectively. Suppose an event `Derived` is derived from `Parent` by adding additional data fields. These new data fields are for passing information into and out of a handler.

When a `Derived` event is raised and caught by a handler bound to `Parent`, it is being treated as a `Parent` event within the handler and the additional data fields cannot be accessed within the handler. Consequently, after the handler terminates normally, some data fields in $\texttt{Derived}_{\texttt{out}}$ may have uninitialized values.

This problem of invalid values can also happen within a class hierarchy during a dynamic dispatch[16]. An example is to initialize an object with its ancestor's constructor. The solution in the class hierarchy is to redefine or override inherited methods[17]. The particular choice of a method in a dynamic dispatch is solely determined by the object to which the method applies. Consequently, if a derived class overrides one of its parent's methods, the overridden method can never be applied to an object of the derived class in a dynamic dispatch. A correct implementation of the overriding method should eliminate all invalid values.

In abnormal event handling, the abnormal event being raised is merely one of the determining factors to select a handler. In particular, a handler for `Parent` can be chosen over one for `Derived` to handle a `Derived` event. As there is always the potential of using a less specific handler, the programmer is responsible for checking the validity of any information coming from a handler.

Apparently, the problem affects the resuming model more than the terminating model because after a resuming handler returns, control flow returns to a context where the event just handled is precisely what was raised, rather than what was caught, i.e., the specific raised event rather than the caught parent event. In the

[16]The virtual method in C++ is a dynamic dispatch facility

[17]It is irrelevant whether the override mechanism uses covariance or contravariance, or even both as suggested by Giuseppe[5] as he claims covariance and contravariance are not antagonistic.

terminating model, control flow never returns to the context of the specific event, and hence, a causal programmer is less likely to make a mistake.

2.14 Summary

Raising, propagating and handling an abnormal event are the three core steps in an AEHM as a flow control mechanism. For safety sake, an AEHM should provide two raising mechanisms: throwing and signalling. There are two useful handling models: terminating and resumption. Handlers should be partitioned with respect to the handling models to provide a better abstraction. Abnormal event parameters, homogeneous derivation of abnormal events and conditional handling are features to improve programmability and extensibility.

An AEHM in a concurrent environment must provide some blocking facilities to solve the reentrant problem. Hierarchical blocking is the best in terms of extensibility and programmability.

The new propagation mechanism I propose solves the recursive signalling problem and is better than existing propagation mechanisms. Indeed, the new propagation mechanism gives a new flow control mechanism.

Some issues remain open. One is the delivery order of multiple pending asynchronous events. Another is on the abnormal event hierarchy. It is still uncertain whether one dual event hierarchy is better than multiple hierarchies in practice.

Chapter 3

Using abnormal events in μC++

This chapter shows how to use the μC++ abnormal event features. Some knowledge about C++ exception handling mechanism, mostly its syntax is assumed (but prior examples have informally discussed almost all of the syntax). A brief overview about the available features is given in section 3.1. Sections 3.2 to 3.5 cover the new syntactic elements for using μC++ abnormal events. Topics include defining new abnormal events, raising an event, writing handlers and using asynchronous abnormal events. A summary is given at the end.

3.1 Overview of μC++ AEHM

μC++ supports the declaration of throw-only, signal-only and dual events. Events are implemented as class objects, just like C++ exceptions; abnormal event parameters are encapsulated inside a class object. Derived events are defined through public class inheritance. Both throwing and signalling are supported. μC++ adopts the propagation mechanism described in section 2.11.3, so recursive signalling is prevented. Resuming handlers are merely C++ routines of some particular signatures because of the lack of nested routines in C++ and the GNU compiler. μC++ also supports implicit polling and hierarchical blocking of asynchronous abnormal events.

The memory management of abnormal events in μC++ is identical to managing C++ exceptions, regardless of whether the event is synchronous or asynchronous. In particular, a programmer is responsible for freeing the heap memory occupied by an event object. Furthermore, the GNU compiler always creates a copy[1] of a thrown event in the heap and does not destroy it automatically. Therefore, a terminating handler generally has to destroy the caught event to prevent memory leaks. However, it is unnecessary, and probably dangerous, to destroy a caught signalled event in a resuming handler because after the resuming handler handles the signalled event, the execution returns to a scope in which the event is still visible. If an event is raised asynchronously, the μC++ kernel copies the event so that the programmer can safely destroy the original copy of the event.

The μC++ kernel implicitly polls for asynchronous events when an execution is about to resume after blocking or after a context switching operation. Section 3.5.1 shows that this is very important.

Generally speaking, an unhandled thrown event causes the termination of the faulting execution; an unhandled signalled event is thrown if it is a dual event; otherwise, it is ignored.

[1]Every abnormal event is μC++ must have a public copy constructor.

3.1.1 Missing desirable features

Since μC++ relies on the exception facilities in the GNU compiler[2], some desirable features are not available. In particular, C++ considers all eligible handlers in a handler clause (catch clause) are equally specific, and the one closest to the keyword `try` is chosen. As a result, μC++ inherits this behaviour. There is no conditional handling facility in μC++ either because it is missing in C++ and the GNU compiler.

Currently, asynchronous events are delivered in a FIFO order. Events currently blocked from delivery do not prevent events arrived later from being delivered. User-defined default handlers for thrown events will be a future addition to the system because the `rtti` facility in the GNU compiler is still in beta stage of development.

3.2 Defining abnormal Events

μC++ provides additional class descriptors to specify the properties of a new class. These class descriptors can be used wherever the C++ keyword `class` can appear. An example is `uCoroutine` for defining coroutines. Indeed, `class` is the only available class descriptor in C++.

Three new class descriptors `uDualEvent`, `uThrowEvent` and `uRaiseEvent` are introduced for defining new abnormal events. A dual event must be derived (or publicly inherited) from `uAEHM::uDualClass` or one of its derived events to ensure homogeneous derivation. If no parent is specified, `uAEHM::uDualClass` is chosen as the default. Similarly, throw-only events must be derived from `uAEHM::uThrowClass` and signal-only events from `uAEHM::uRaiseClass`. No event can be derived from

[2]A μC++ program is translated to C++ code and then compiled by the GNU compiler.

more than one event.

For every abnormal event declared, e.g. `SomeEvent`, an initialization statement of the form

```
uInitEvent(SomeEvent);
```

is required to force the GNU compiler to define and allocate static storage for information associated with a event. The initialization statement for every event can be invoked exactly once in the entire program or the GNU compiler is confused by duplicated definition.

C++ also has non-public inheritance and μC++ has enriched the C++ basic class objects with the addition of monitors, coroutines, tasks and abnormal events. Though C++ supports multiple inheritance, not all forms of multiple inheritance have acceptable semantics[1]. The following explains what other forms of inheritance is acceptable involving abnormal events.

The default resuming handler for `uAEHM::uDualClass` is to throw the event, and that for `uAEHM::uRaiseClass` simply returns the value `uAEHM::HANDLED` as if the signalled event is handled by a non-default handler. Both are implemented as a virtual method of the following signautre:

```
virtual uAEHM::uRaiseReturn uDefaultResume() const;
```

A user-defined default resuming handler is implemented by overriding the virtual method. The keyword `const` in part of the signature as it is inC++. The return type `uAEHM::uRaiseReturn` is further described in section 3.4 when discussing handlers. As mentioned, default terminating handler is not supported at the moment.

3.2.1 Events inheriting from other object class

An abnormal event is not allowed to publicly inherit from non-event classes. Public derivation of events is for building an event hierarchy, and the restriction on public inheritance should enhance the distinction between class hierarchy and event hierarchy. Furthermore, the operations on events and non-event objects are usually defined differently. Handlers are for events and methods (or entries) for classes. The former is a dynamic choice; the latter static. It appears that keeping events and non-event objects separate is reasonable.

Non-public inheritance is sharing of implementation. I cannot imagine any circumstances where an event needs to inherit from a coroutine or a task. Why would an abnormal event need context switching or a thread to carry out computation after all? And what should happen when a hybrid of execution and event encounters another abnormal condition? Consequently, any form of inheritance from a coroutine or a task by an abnormal event is rejected.

Though the same abnormal condition may be encountered by different executions, it does not mean that an abnormal event can or should be a shared resource. For example, arithmetic overflow can be encountered by different executions but each arithmetic overflow is an independent entity. Hence, there is no race condition for events, and therefore, non-public inheritance from a mutex object by an abnormal event should not be useful. As a result, it is rejected.

Finally, there is the non-public inheritance from an ordinary class object. This form is the only acceptable inheritance by abnormal events from non-events. An example for using such inheritance is a set of abnormal events using similar logging routines. An ordinary class can implement the logging functionalities and consequently be reused by the events.

3.2.2 Non-event classes inheriting from events

As mentioned, events should be kept distinct from non-events objects. Therefore, non-event classes cannot publicly inherit from events.

In addition, operations defined for events like raising an event should not be shared by non-event objects. Hence, non-public inheritance from events by non-events is rejected as well. However, I do not mean that events and non-events cannot share code. Rather, the shared code should be implemented as an ordinary class and then inherited by events and non-events.

3.3 Raising an abnormal event

There are two propagation mechanisms — throwing and signalling. A programmer specifies how a event is propagated in the faulting execution with the different raise statements. The following are the acceptable forms for raising an event:

```
uThrow throwable();
uThrow throwable() uAt target;
uRaise raiseable();
uRaise raiseable() uAt target;
```

The `uThrow` statement is for throwing an event. With the optional "`uAt target`", the event is thrown at coroutine or task `target` asynchronously by the executing coroutine. The `uRaise` statement is for signalling an event. The keyword `uRaise` is used because `uSignal` in μC++ is associated with an operation for condition variables.

The C++ `throw` statement truncates events. In other words, if `x` is declared as an object of class `X`, the event raised by

```
throw x;
```

can only be caught by a handler capable of catching `X`. It is possible that the object `x` refers to an object of class `Y` derived from `X`, but a handler catching `Y` cannot catch the thrown `x`. Neither `uThrow` nor `uRaise` in μC++ truncate events, i.e., the event actually raised at runtime can be a derived event of the one specified in the statement.

To re-throw an event inside a terminating handler, it is necessary to use the following to avoid memory leaks:

```
uThrow;
```

which is only valid in the context of a terminating handler. However, no similar form exists for signalling because first, it is unnecessary as the memory leak does not exist, and second, it is impossible to have a syntactic construct exclusively for resuming handlers but not for functions.

A resuming handler should return a value of type `uAEHM::uRaiseReturn` if it terminates normally. This return value becomes the result of the synchronous form of the `uRaise` statement. Indeed, the synchronous form of `uRaise` can be used as an expression returning a value of type `uAEHM::uRaiseReturn`. The proper use of this return value is discussed in section 3.4.

The synchronous `uThrow` statement does not have a return value because the current scope terminates immediately when executing the statement, so there is no place to return a value to. The asynchronous `uRaise` and `uThrow` do not have a return value because the handler for the event is invoked asynchronously.

3.4 Handlers

Terminating handlers in μC++ are identical to those in C++ in syntax and behaviour. A terminating handler is defined with the catch clause as in:

```
try {
    // statements to be guarded
}
catch (uAEHM::uDualClass &e) {
    // handling statements
}
```

A resuming handler in μC++ for abnormal event `AE` is a function with either of the following signatures:

```
uAEHM::uRaiseReturn (*)(AE &)
uAEHM::uRaiseReturn (*)(AE &, ARG &)
```

The second allows additional arguments (which are bundled into an object of class `ARG`[3]) to be passed to the handler when handling an event. The former handler does not take any arguments when handling the event.

[3] Class `ARG` must be publicly derived from `uAEHM::uClosure` because of limited support for templates in the GNU compiler.

The return value from a handler belongs to the enumeration type:

```
enum uAEHM::uRaiseReturn {NOT_HANDLED, PARTIALLY_HANDLED, HANDLED}
```

By convention, `NOT_HANDLED` is returned when a resuming handler is not found. `HANDLED` is returned after a handler handles an event. In case a handler is not found when re-signalling an event, the active handler can return `PARTIALLY_HANDLED` instead of the other two values to distinguish the different cases.

If a handler chosen to handle a signalled dual event returns `NOT_HANDLED`, the runtime system considers that the faulting execution fails to handle the event. Subsequently, the event is thrown.

It is always recommended to catch the event by reference in both types of handlers. Otherwise, the event caught by a handler is a truncated copy of the event raised, and causing a loss in specificity when re-raising the event in the handler. The additional copying of a thrown event can result in memory leaks.

3.4.1 Binding handlers to a block

μC++ extends the try block in C++ to set up handlers for a (guarded) block. A terminating handler is specified with a catch clause as it is in C++ . A resuming handler in μC++ is a function as described previously. The following is an example of binding resuming handlers to a block:

```
try <AE1,h1,arg1><AE2,h2><AE3><AE4,h4,arg4> {
   // block protected by handlers
}
```

In general, the `try` block in μC++ behaves like a template and can take an arbitrary number of template arguments. The following three forms of template arguments are allowed:

1. `<AE1,h1,arg1>`

2. `<AE2,h2>`

3. `<AE3>`

The first one specifies that handler (or function) `h1` handles signalled event `AE1`. Because nested function is not available, `arg1` (of class `ARG1`) is used in the first form for passing local information to handler `h1` when handling the event. Therefore, the signature of `h1` must be:

```
uAEHM::uRaiseReturn h1(AE1 &, ARG1 &)
```

The difference between the second and the first is that the handler `h2` for event `AE2` does not take additional arguments. Therefore, its signature must be:

```
uAEHM::uRaiseReturn h2(AE2 &)
```

The third one specifies that the event `AE3` is handled by a handler that simply returns the value `uAEHM::HANDLED`. This conveniently ignores certain signalled events.

Type checking is performed on the first two forms, but not the last one, to ensure a proper handler is chosen to handle the designated abnormal event. In general, the given handler should follow contravariance as a handler is chosen to handle a descendent but not an ascendent of an event[4].

[4]Currently, a warning is issued by the GNU compiler when a resuming handler satisfies controvariance. It is an error if the handler violates both controvariance and contravariance.

3.5 Delivery of asynchronous events

μC++ allows a programmer to turn on (and off) the delivery of asynchronous events. An event is blocked if its delivery is turned off. Note that the facility only affects asynchronous events. Initially, the delivery of every asynchronous event is turned off in an execution, so that a programmer can set up handlers for various asynchronous events. The constructs used for turning on and off asynchronous delivery are the `uEnable` block and the `uDisable` block:

```
uEnable <AE1><AE2> {            uDisable <AE1><AE2> {
  // code in uEnable block        // code in uDisable block
}                               }
```

The template arguments of the `uEnable` block (or `uDisable` block) specify what events are allowed to be delivered (or blocked). Specifying no template arguments is a shorthand for specifying all abnormal events. Though an asynchronous event being delivered may match with more than one event specified in the closest `uEnable` (or `uDisable`) block, it is unnecessary to define how the kernel matches it with the specified events because the asynchronous event is either enabled or disabled regardless of which event it is matched with.

An execution can enter different `uEnable` and `uDisable` blocks, and as a result, alter the delivery of an event. When an execution enters a `uEnable` block, the events specified in the block become deliverable, even if the events are currently blocked. Similarly, an execution entering a `uDisable` block blocks the delivery of the specified events.

Upon exiting a `uEnable` or `uDisable` block, the execution restores the delivery of various asynchronous events to the state before entering the block.

3.5.1 Polling for asynchronous events

Section 2.10.1 on page 57 states that there are explicit and implicit polling. A μC++ programmer can explicitly poll for asynchronous abnormal event by calling the function:

```
void uAEHM::uDeliverEvents();
```

Blocking a thread is a basic part of synchronization and mutual exclusion. When a synchronization or a mutual exclusive operation cannot complete normally, an asynchronous event is sent as a notice to the blocked execution. An asynchronous event can also be used for communicating abnormal conditions encountered among executions. In general, it is unsafe to delay the handling of these abnormal events. Therefore, implicit polling for asynchronous event when an execution wakes up or becomes active provides a chance to handle these asynchronous events immediately, unless the event is disabled.

Currently, the μC++ kernel ensures that an execution implicitly polls for asynchronous events after it wakes up from a blocking operation, and when it becomes active after a context switch. Asynchronous abnormal events can be raised for abnormal conditions encountered in synchronization and mutual exclusion. The next chapter studies these abnormal conditions, and a few others in the μC++ kernel. For safety sake, the runtime system turns on the delivery of asynchronous events raised as a result of these abnormal conditions, preventing the user from blocking the event.

3.6 Summary

This chapter describes the features and the new syntax of the μC++ AEHM. Abnormal events are class objects that can be thrown or signalled. There are three kinds of abnormal events: `uDualEvent`, `uThrowEvent` and `uRaiseEvent`. Each has its own event hierarchy. These hierarchies are built by publicly inheriting from another event of the same kind. The only other acceptable form of inheritance involving an abnormal event in μC++ is non-public inheritance from a normal class (i.e., specified by the class descriptor `class`) by an event.

μC++ supports throwing and signalling abnormal events, synchronously or asynchronously. A terminating handler is defined by the `catch` phrase as it is in C++. A resuming handler is defined as a function with some restrictions on its signature. The C++ `try` block is extended to a properly type-checked, template-like facility in order to support the resumption handling model.

The delivery of asynchronous events may be additionally controlled by the `uEnable` and `uDisable` blocks. Both implicit and explicit polling are supported.

Chapter 4

Programming with μC++ abnormal events

This chapter describes how abnormal events are used in the μC++ kernel to deal with abnormal conditions. Previously, these conditions either abort the program or are ignored. A closer look at certain features in μC++ is given as well so that the abnormal conditions in the μC++ kernel can be subsequently identified.

First, section 4.1 provides some guide lines for better use of abnormal events. Then, section 4.2 discusses synchronization in concurrent programming. The μC++ synchronization mechanism *rendezvous* is studied in detail in section 4.3. The majority of this section is to illustrate how *condition variables* may affect the rendezvous mechanism. The abnormal conditions in the μC++ kernel are listed and discussed in section 4.4. Some of these abnormal conditions do not exist in other languages because coroutines and context switching operations are missing in these languages. An abnormal event hierarchy is also given. Section 4.5 has two examples to show how abnormal events can be used, followed by a summary section. The first example is a maze search program. The other is an iterator for tree transversal. The implementation uses coroutines. A summary is given at the end as well.

4.1 General guide lines

Like many other programming features, the AEHM aims to make certain programming tasks easier and improve the overall quality of a program. Indeed, choosing abnormal events over other available flow control mechanisms is a tradeoff. For example, a programmer may decide to use abnormal events over some conditional statement for clarity. This decision may sacrifice runtime efficiency and memory space. It is extremely difficult to estimate the tradeoff without knowing the internal implementation of the compiler/interpreter being used. In other words, universal, crisp criteria for making a decision do not exist. Nevertheless, some important guide lines are given below to avoid abusing abnormal events.

First, using abnormal events when writing a library to indicate rare conditions can prevent the library client from ignoring them if the client "forgets" to set up a handler. Hence abnormal events can improve safety and robustness.

Second, using abnormal events can improve clarity and maintainability over techniques like status return values and status flag where normal flow and abnormal flow are mixed together throughout a block. The abnormal event technique not only separates clearly the normal flow in guarded regions from the abnormal flow in handlers, but also avoids mixing the normal return values with the error return values. The abnormal event technique accommodates changes better, too.

Third, using abnormal events to indicate conditions that happen rarely at runtime. The reason is two fold. First, the normal flow of the program represents what should happen most of the time, so programmers can easily understand the functionality of a code segment. The abnormal flow then represents subtle details to handle these rare conditions, such as boundary conditions. Second, the propagation mechanism requires a search for the handler and is usually expensive. Part of the cost is a result of the dynamic choice of a handler. (See section 2.4.1 for why the dynamic choice is essential.) Nonetheless, the dynamic choice of a handler is less understandable than a normal routine call. If abnormal events are raised frequently, there is a high runtime cost and the control flow of the program can become less understandable.

4.1.1 Using a thrown event

Typical use of a thrown event is for *graceful* termination of an operation or even an execution. A termination is graceful if the termination triggers a sequence of clean up actions in the execution context. Examples of *abrupt* (or non-graceful) termination include the `abort` function in `C` and the `kill -9` command in Unix. Graceful termination is more important in a concurrent environment because one execution can terminate while others can continue. The terminating execution, or operation, must be given a chance to release any shared resources it has claimed (the clean up action) in order to maintain the integrity of the whole environment. For example, deadlock is potentially a rare condition and a thrown event can force graceful termination of a blocked operation, consequently leading to the release of some shared resources and breaking of the deadlock.

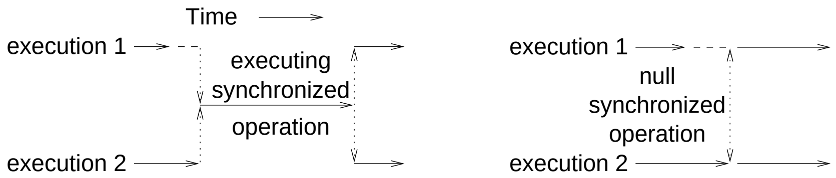
4.1.2 Using a signalled event

A signalled event causes the faulting execution to do additional computation in the form of a resuming handler that it does not do normally. The additional computation may modify the state of the execution, which can be considered as error recovery. Alternatively, it may cause information about the execution to be gathered and saved without effectively modifying the execution state.

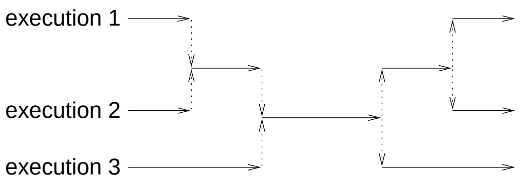
4.1.3 Using a dual event

A dual event is either thrown or signalled, and hence can be used as a thrown event as well as a signalled event. What makes dual events unique is that, unlike throw-only and signal-only events, they can be thrown and signalled. In situations without a clear choice of termination or resumption, a dual event can be signalled initially so that resumption is feasible to avoid loss of local information in the termination model. If no resumption handler can handle the event, the same event can be thrown.

One possibility of such use is in real-time applications. Missing a real-time constrain, say an execution cannot finish before a deadline, is considered an abnormal condition. For some applications, the constrain violation can result in termination. Other applications can modify some internal parameters to make its execution faster by sacrificing the quality of the solution, or by acquiring more computing resources making the resumption model appropriate. The dual event is ideal for this kind of abnormal condition.



(a) synchronization session (b) synchronization point



(c) nested synchronization

Figure 4.1: Synchronization

4.2 Synchronized operation

Any concurrent programming environment must support mutual exclusive and synchronizing threads of execution.

When two threads synchronize, an operation is carried out. This *synchronized operation* may be a null operation. When it terminates, the two threads continue to execute independently and in parallel. Usually, one thread has to wait for another before a synchronization. A *synchronization session* is the period of time the two executions[1] synchronize. It is a *synchronization point* if the synchronized operation is null. See figures 4.1(a) and 4.1(b). During a synchronized session, an additional synchronization requirement may have to be met before the execution of the synchronized operation can continue. Therefore, synchronized operations

[1]Precisely, I should say the active execution of the threads. Two executions bound to the same thread cannot synchronize because at most one of them is active at any time.

can be nested as shown in figure 4.1(c). The synchronizations in the figure are symmetric as the executions have identical roles in the synchronization.

4.2.1 Rendezvous

Synchronization often involves two executions performing two different roles, as in the client-server model. Therefore, rather than providing an abstraction for symmetric synchronization, Ada 83, Ada 95[2] and μC++ provide an asymmetric synchronization facility known as *rendezvous*. The synchronized operation is implemented as an *entry*, a routine-like element. A rendezvous begins only when one of the executions (the *caller*) invokes an entry, and the other (the *acceptor*) accepts the entry invocation. The acceptor may execute a sequence of *post-synchronization* statements. Invoking or accepting an entry by an accepted entry may create another rendezvous. Therefore, rendezvous can be nested many levels.

A rendezvous *terminates* when exiting the accepted entry. Rather than executing to completion, a rendezvous may terminate abnormally by a thrown event as well. It is desirable to inform both the caller and the acceptor about the abnormal termination.

Because Ada does not provide facilities to asynchronously raise an abnormal event, informing the caller and the acceptor about an abnormal termination of a rendezvous is a built-in facility in the language.

[2]For convenience, I denote the two standards of Ada as Ada 83 and Ada 95. The term *Ada* refers to the whole Ada language family.

4.3 μC++ rendezvous

μC++ also has rendezvous and the synchronized operation is implemented as an entry, but its rendezvous mechanism and that of Ada family have subtle and fundamental differences.

An object in μC++ not only has an object state, but may also have an execution state and a thread. It may also require mutually exclusive access. The properties of the μC++ objects are described in section 1.3, and μC++ uses an entry to abstract various requirements imposed by the kind of an object. When discussing rendezvous, it is sufficient to classify the entry into two categories: blocking and non-blocking entries. A blocking entry avoids race conditions by blocking its caller thread when the mutex object is locked by another thread. A non-blocking entry is like a normal routine and never blocks its caller.

μC++ allows only blocking entries to be accepted. The acceptor must be the execution locking the mutex object. Besides getting accepted, an invoked blocking entry can begin executing when no other thread is locking the object. Whether an entry invocation proceeds with or without an acceptor is generally a runtime phenomenon. On the contrary, the Ada language defines what can be accepted and what not at compile time, reflecting the conceptual distinction between mutual exclusion and synchronization.

Obviously, the μC++ design is more flexible as rendezvous can help controlling the order of entry execution of a mutex object. Indeed, rendezvous in μC++ is a control mechanism allowing the acceptor to select a particular blocking entry. Upon accepting an entry, the acceptor passes the exclusive right to access a mutex object to the caller, and subsequently, the caller can proceed.

4.3.1 Condition variables in μC++

μC++ provides condition variables to suspend a thread executing a blocking entry. Since an accepted entry must be a blocking entry, the μC++ orthogonal design allows suspending a thread executing an accepted entry as well.

A condition variable in μC++ is always associated with one and only one mutex object — the *owner* of the condition variable. μC++ has two statements operating on condition variables that affect how n entry is executed:

- `uWait` blocks the current thread of control and forces it to release the lock on the mutex object owning the condition variable.
- `uSignal` removes a blocked thread from a condition variable, if there is one, and pushes it onto an internal stack of the mutex object. The pushed thread gains control of the mutex object only when it is chosen by the internal scheduler after the mutex object becomes unlocked. `uSignal` has no effect if no thread is blocked on a condition variable.

Although the condition variable is a common feature in concurrent languages and libraries, no work has shown how condition variables interact with rendezvous because the two facilities are usually not available together. The rest of this section discusses how condition variable affects the control flow of the caller and acceptor of a rendezvous.

4.3.2 Condition variable and rendezvous

Upon establishing a rendezvous, the μC++ kernel pushes the acceptor onto the internal stack of the mutex object, and forces the acceptor to relinquish the mutex

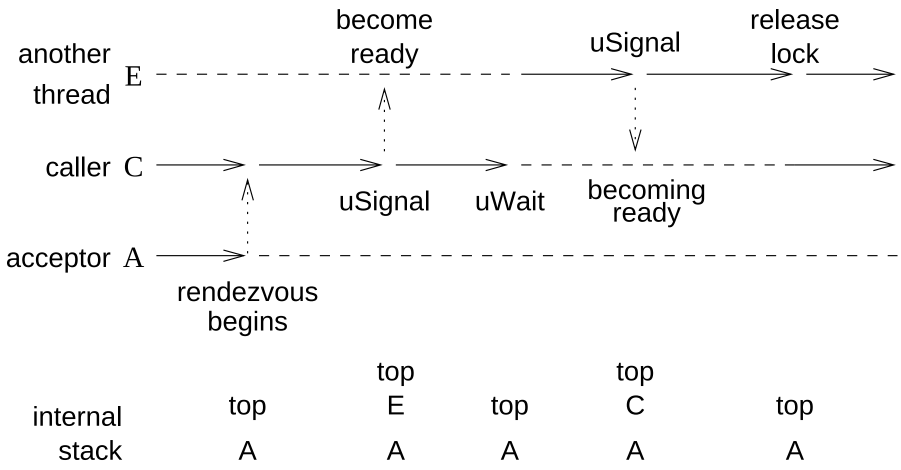


Figure 4.2: An example of `uWait` and `uSignal`

object to the caller, giving the caller exclusive access to the mutex object. While executing the accepted entry, the caller can `uSignal` any condition variable in the mutex object, possibly pushing more threads onto the internal stack. When the caller releases the mutex object by executing a `uWait` statement or terminating the entry invocation, the internal scheduler chooses the thread on the top of the internal stack in the mutex object to resume executing the previously blocked entry. Therefore, the acceptor may not be restarted immediately. Furthermore, if another thread is chosen, this thread can wake up other threads (possibly the caller) with `uSignal` as illustrated in figure 4.2. As a result, the caller resumes before the acceptor.

An execution can `uWait` on a condition variable whenever the current thread locks the owner of the condition variable. However, the `uWait` only suspends the thread executing the blocking entry. The acceptor of the entry, if there is one, is not suspended. Though it is possible for the internal schedular to pick the acceptor, the `uWait` mechanism does not guarantee the acceptor to resume. It is even possible that the caller wakes up and exits the accepted entry before the acceptor proceeds to the post-synchronization statements. It all depends on which thread gets the lock of the mutex object at different time, as illustrated in figure 4.3. The thick solid line is used when the caller of a rendezvous owns the mutex object; the thin solid line for the acceptor. The dotted line does not specify who owns the mutex object; the owner can possibly be the caller, but not the acceptor.

Obviously, the `uWait` mechanism should be considered as suspending the current thread but never suspending a rendezvous[3]. Any mechanism that suspends a rendezvous should suspend both the caller and the acceptor, plus releasing the lock of the mutex object. Without releasing the lock, the mechanism is not significantly different from nested rendezvous.

There is a subtle point for `uWait` suspending a blocking entry. μC++ allows a thread to enter a mutex object multiple times once it has acquired the mutex lock. Figure 4.4 shows an execution having invoked blocking entries of object `A` three times. Note that after a thread has entered a mutex object, any subsequent entry invocation of the same object by the thread cannot be accepted, because an accepted entry requires a different thread executing an accept statement in the mutex object. Now if the thread in figure 4.4 `uWaits` on a condition variable of mutex object `A`, the lock of object `A` becomes free and the acceptor of `entry 1` of `A` may resume. The acceptor of `entry 1` of object `B` cannot resume because the lock of `B` is not free.

[3]Using "rendezvous" to refer to the execution of an accepted entry in μC++ may be confusing for those familiar with Ada rendezvous mechanism, especially with the ability to suspend the caller but not the acceptor.

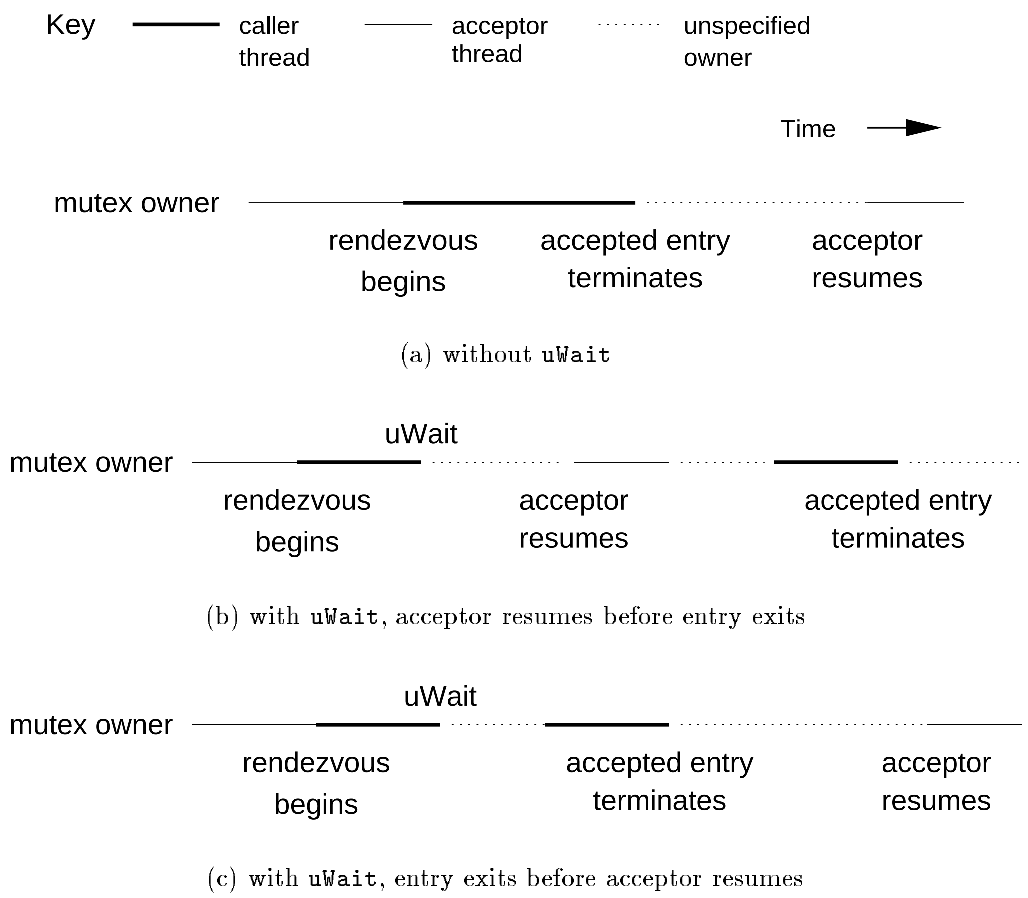


Figure 4.3: Ownership of a mutex object when executing an accepted entry

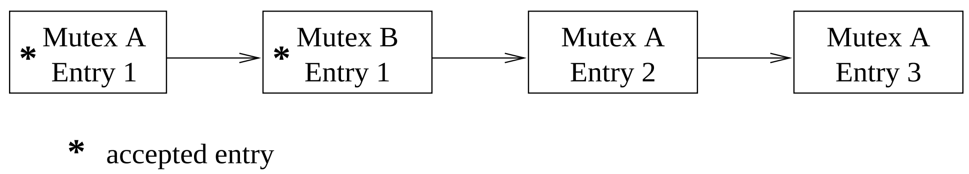


Figure 4.4: Entering a mutex object multiple times

4.3.3 Abnormal termination of an accepted entry in μC++

The `uWait` and `uSignal` makes it difficult to determine whether the acceptor should be informed about the abnormal termination of an entry because of all the possible scenarios shown in figure 4.3. However, unlike Ada, μC++ allows throwing and signalling of an asynchronous abnormal event. A μC++ programmer can easily signal or throw an asynchronous event at the acceptor by using the macro

`uRendezvousAcceptor`

to determine the acceptor of a blocking entry. The marco returns a pointer to an execution (`uBaseCoroutine *`). It returns the pointer to the acceptor if one exists and remains blocked as a result of accepting the entry. Otherwise, the macro returns `NULL`. The macro is applicable only inside a blocking entry. In figure 4.4, the macro returns `NULL` in `entry 2` and `entry 3` because there is no acceptor.

4.4 Abnormal conditions in the μC++ kernel

This section discusses some important abnormal conditions in the μC++ kernel based on the principles outlined in section 4.1. The scope of the discussion is limited to a concurrent programming context. Abnormal conditions like `cannot open file` in file operations are not discussed because of previous work on abnormal events in sequential programming.

μC++ uses entries to hide context switching, mutual exclusion and synchronization inside an object from users. Abnormal conditions encountered as a result of these operations in a concurrent environment are the focus of this section. Generally speaking, if an entry invocation encounters an abnormal condition, an abnormal event is raised, similar to the way an event is raised inside an object method in a sequential language.

It is important to realize that for objects that have an execution state, if the execution terminates, the object is still accessible and it retains its mutex property. This semantics is a result of the μC++ orthogonal design. Consequently, invoking an entry of a terminated task is functionally equivalent to invoking an entry of a mutex object. Only when an object is destroyed, does access to the object become erroneous. Currently, accessing a destroyed object results in undefined behaviour in μC++ because the programmer is responsible for memory management issues, which is true for all C++ objects.

The abnormal conditions are discussed from section 4.4.1 to section 4.4.3. Section 4.4.4 covers what events are raised in these conditions, with the event hierarchy for the μC++ kernel.

4.4.1 Abnormal conditions in context switching

Context switching refers to a change in the execution binding of a thread. A context switch is executed sequentially as only one thread is available. μC++ context switching facilities are the `uSuspend` and the `uResume` statements.

`uResume` is for resuming an inactive execution. However, it is possible that the execution is terminated when this happens, and hence, an event is thrown to abnormally terminate the operation. Similarly, suspending the current execution by executing `uSuspend` may cause a context switch back to a terminated execution, causing the abnormal termination of the operation.

Occasionally, an execution wants to communicate an abnormal condition to another execution. In particular, it appears desirable that the one that most recently resumes the current execution. Consider a coroutine that implements a server. A request is made when invoking one of its entry routine which results in a context switch to the server. If the server coroutine terminates abnormally and consequently fails to satisfies the most recent request, it is reasonable to inform the client, i.e, the coroutine that most recently resumes the server.

As a result, μC++ provides programmers a method to determine the last execution that resumed a coroutine:

```
uBaseCoroutine& uBaseCoroutine::uLastResume();
```

The entry returns `NULL` if there has not been a resumer.

4.4.2 Abnormal termination of an execution

An execution may terminate abnormally upon a thrown event. One of the goals of using abnormal events is to prevent abnormal conditions from being ignored easily. In a concurrent environment, this requires informing other executions.

If the execution is a coroutine, a dual event is signalled asynchronously at the execution which starts[4] the terminating coroutine. A signalled dual event is used because the faulting execution may want to delay or ignore the condition, and can continue the propagation if no explicit action is specified for the dual event. The event is asynchronous simply because the condition originates from an external execution. The starter is chosen over the last resumer for the following reasons:

[4]A coroutine starts executing when another execution does the first resume of a coroutine.

- The start relation of coroutines is hierarchical but the resume relationship is potentially cyclic. Using the last resumer may lead to incomprehensible behaviour because the resumer can be a terminated coroutine.

- If a thrown event cannot be handled by a coroutine, the programmer can choose to propagate the event at the last resumer using a terminating handler like:

```
void main() { // the main body of a coroutine
    try {
        ...
    }
    catch(all_event &e) {
        uThrow e uAt *uLastResume(); delete &e;
    }
}   // end main
```

  and let the coroutine terminate normally.

- When a coroutine terminates normally, the default semantics of μC++ also chooses to resume the starter. Extending this to the abnormal termination seems reasonable.

One would expect a similar design to handle abnormal termination of a task. However, this is not the case because μC++ does not impose any hierarchical terminating sequence of tasks. There seems no reasonable choice of an execution in which the propagation of the event should continue. Furthermore, when a task terminates normally, its thread is dead and no more execution is done by the thread. Extending this to the abnormal case is to ignore the thrown event at the end, which is undesirable. Consequently, the μC++ kernel simply aborts the whole program when a task terminates abnormally.

4.4.3 Abnormal conditions for invoking a blocking entry

Besides abnormal conditions raised while an execution is running, i.e., actively executing statements inside an entry, invoking a blocking entry may lead to additional abnormal conditions absent in sequential computation.

First, a mutex object can be destroyed while threads are blocked waiting to access the mutex object after invoking a blocking entry. These threads can be blocked on the entry queues, in a condition variable or the internal stack of the mutex object. The blocked entry invocations should not be allowed to continue. Therefore, an asynchronous event is thrown at the executions executing these entries by the execution destroying the mutex object. Note the μC++ kernel always enables the delivery of this event, so it is impossible to prevent the event from being disabled at the faulting execution.

As mentioned, an entry invocation can be a rendezvous. For the caller, the behaviour of invoking a blocking entry is the same whether the entry is accepted or not. The programmer can signal a dual event at the acceptor if the entry is accepted. Using a dual event with the signal mechanism makes resumption possible.

A rare anomaly can happen. Both the acceptor and the caller of an active rendezvous can be blocked in an mutex object as a result of nested rendezvous. While they remains blocked, another execution can destroy the mutex object. Consequently, the caller and the acceptor receive an asynchronous thrown event. The caller, upon the abnormal termination of the accepted entry, signals another event at the acceptor. This semantics is undesirable because there are two asynchronous events arrives at the acceptor as a result of the "premature" destruction of the mutex object. The programmer must be aware of this situation.

uAEHM::uDualClass
- uKernelFailure
  the root of all kernel events
  - uSerial::uFailure
    general abnormal condition in a blocking entry
    - uSerial::uEntryFailure
      entry fails while being blocked at entry or suspended in internal stack
    - uCondition::uSuspendedFailure
      entry suspended in a condition variable terminates abnormally
  - BaseCoroutine::uFailure
    general abnormal condition in an execution
    - uBaseCoroutine::uTerminated
      context switching to a terminated execution or
      abnormal termination of a coroutine

Figure 4.5: μC++ abnormal event hierarchy

4.4.4 μC++ abnormal event hierarchy

Previous sections discuss abnormal conditions in the μC++ kernel and this section states what abnormal events are raised for these abnormal conditions. Figure 4.5 shows the abnormal event hierarchy defined for the μC++ kernel. The event `uBaseCoroutine::uTerminated` is signalled asynchrounsly at the starter of a coroutine that terminates on a thrown event. All other abnormal conditions result in a thrown event because the current operation is not resumable.

The kernel abnormal events are all dual events because dual events are more flexible. However, μC++ allows defining throw-only and signal-only events, partly because it is far from certain whether throw-only and signal-only events are useless. More importantly, the presence of throw-only and signal-only events allow a programmer to experiment with with abnormal flow control in their application.

4.5 Abnormal events as a flow control mechanism

This section provides two examples of using abnormal events more as a flow control mechanism than an error handling mechanism. The examples are single-threaded because the focus is to illustrate how the control flow changes in an execution. The first example illustrates how signalled events can be used, followed by one showing how to use thrown events with multiple coroutines.

4.5.1 Searching a maze

The program fragment in figure 4.6 is from a maze search program. Some of the details of `class maze` are omitted. The program creates a maze and starts searching an exit path from location `(5,10)`. It searches around that position and if the exit is found, an abnormal event `ShowPath` is signalled. The resuming handler set up at each recursive call re-signals the event before printing out a location. Abstractly, the signalling of `ShowPath` causes the execution to print out the path from the starting position to the current position. The program terminates on a thrown abnormal event `Quit`.

The program does not use return values to indicate the success or failure of the search. Nor does it use the heap to keep track of its current search path. It takes advantage of the closure implemented to support resuming handlers and nested routines. Unlike recursive routine call, the re-signalling does not cause infinite recursion with the new propagation mechanism. Another point worth mentioning is that in a concurrent environment, another execution can signal `ShowPath` at an execution finding an exit path and create a snapshot of the current search path.

```
#include<uC++.h>
#include<iostream.h>

uRaiseEvent ShowPath {};
uThrowEvent Quit {};

struct Position : public uAEHM::uClosure {
  int x,y;
  Position(int x, int y) : x(x), y(y) {}
};

uAEHM::uRaiseReturn showPos(ShowPath &event, Position &p) {
  uRaise event;    // resignal the event
  cout << "At grid (" << p.x << ',' << p.y << ")\n";
  return uAEHM::HANDLED;
}

class maze;  // details omitted

void maze::search(int x, int y) {
  Position pos(x,y);
  try <ShowPath, showPos, pos> {
    if targetReached(x,y) {
      signal ShowPath; throw Quit;
    }
    if visited(x,y) return;

    search(x+1,y);   search(x-1,y);  search(x,y+1);  search(x,y-1);
  }   // end try
} // end maze::search

main () {
  maze t;
  try <ShowPath> {
    t.search(5,10);
  }
  catch (Quit) {
    cout << "path found\n";
  }
}
```

Figure 4.6: A maze search program using signalled abnormal events

4.5.2 Inorder enumeration of binary tree

Inorder enumeration of a binary tree retrieves the elements in the binary tree, one element at a time, following the order of inorder traversal. The code in figure 4.7 illustrates how to implement an iterator in μC++ to accomplish this. The emphasis of this implementation is to show that the μC++ coroutine can do what a Sather style iterator[21] does. Indeed, the fundamental ideas of the given solution is from the language Sather.

Each iterator, essentially a μC++ coroutine, for a tree creates an iterator for each of its subtree. When a tree is exhausted, a thrown event is raised at the execution invoking the entry routine `next` of an iterator. The iterator then terminates and no more elements can be retrieved from the exhausted iterator because resuming a terminated coroutine is invalid.

The implementation uses a lot memory as each coroutine has its own stack. Indeed, it is unnecessary to create an iterator for every subtree. Figure 4.8 is a better implementation using less memory as only one coroutine.

In both examples, the only way to notice the main function `uMain::main` the end of the transveral is by a thrown event because all the values in `int` are a legitimate return value for the entry `InTran::next`.

4.6 Summary

This chapter identifies the abnormal conditions that may arise from terminating an execution in a concurrent environment, context switching, and executing a blocking entry. These conditions are chosen because they must be handled in a unique

```
#include<uC++.h>
#include<iostream.h>

class btree {
private:
  struct tnode {
    int key;
    btree *lf;
    btree *rt;
    tnode(int k) : key(k) {
      lf = new btree();
      rt = new btree();
    }

    virtual ~tnode() {
      if (lf) delete lf;
      if (rt) delete rt;
    }
  };  // end tnode

  tnode *node;

public:
  btree() : node(NULL) {};
  ~btree() {
    if (node) delete node;
  }

  bool is_empty() {
     return node == NULL;
  }

  void insert(int x) {
    if (node != NULL) {
      if (x <= node->key)
        node->lf->insert(x);
      else
        node->rt->insert(x);
    }
    else {
      node = new tnode(x);
    }
  }

  uCoroutine InTran {

    btree &root;
    int result;

    uThrowEvent done {
    public:
      done() {}
      ~done() {}
    };

  public:
    InTran(btree &t) : root(t) {}

    int next() {
      if (root.is_empty()) {
        uThrow done();
      }
      uResume;
      return result;
    }

  protected:
    void main();

  };  // end btree::InTran

};

// continue on the following page
```

Figure 4.7: Sather style iterator in μC++

```
void btree::InTran::main() {
  if (root.is_empty()) {
    uThrow done() uAt
      *this->uLastResume();
    uSuspend;
  } else {

    uEnable {
      try {
        InTran lf(*root.node->lf);
        for(;;) {
          result = lf.next();
          uSuspend;
        }
      }
      catch(done &d) {delete &d;}

      result = root.node->key;
      uSuspend;

      try {
        InTran rt(*root.node->rt);
        for(;;) {
          result = rt.next();
          uSuspend;
        }
      }
      catch(done &d) {
        delete &d;
        uThrow done() uAt
          *this->uLastResume();
    }}  // end uEnable
  }  // end if
} // end btree:InTran::main

uInitEvent(btree::InTran::done);

void uMain::main() {
uEnable {
  srand(1000);
  btree t;

  for(int i=0; i< 20; i++) {
    int tmp = rand() % 65536;
    cout << tmp << ' ';
    t.insert(tmp);
  }
  cout << endl << endl;

  btree::InTran iter(t);

  try {
    for(;;) {
      cout << iter.next() << endl;
    }
  }
  catch (btree::InTran::done &d) {
    cout << endl;
    delete &d;
  }
}}  // end uMain::main
```

```
#include<uC++.h>
#include<iostream.h>

class btree {
  // details omitted
};


uCoroutine btree::InTran {
  btree &root;
  int result;
  bool exhausted;

  void traverse(btree &t) {
    if ( t.is_empty() )
      return;

    traverse(*t.node->left);

    result = t.node->key;
    uSuspend;

    traverse(*t.node->right);
  }   // end traverse

public:
  InTran(btree &t) : root(t) {}

  int next() {
    uResume;
    if (exhausted)
      uThrow uAEHM::uThrowClass();
    else
      return result;
  }

  void main() {
    traverse(root);

    // end of transversal at this point
    exhausted = true;
  } // end btree:InTran::main

};  // end uCoroutine btree::InTran


void uMain::main() {

  srand(1000);
  btree t;

  for(int i=0; i< 20; i++) {
    int tmp = rand() % 65536;
    cout << tmp << ' ';
    t.insert(tmp);
  }
  cout << endl << endl;

  InTran iter(t);

  try {
    for(;;) {
      cout << iter.next() << endl;
    }
  }
  catch (uAEHM::uThrowClass &ae) {
    cout << endl;
    delete &ae;
  }
} // end uMain::main
```

Figure 4.8: μC++ implementation of an inorder tree iterator

fashion in μC++ as a result of its orthogonal design and its context switching facility. Indeed, abnormal events are successfully introduced without changing the orthogonal design and semantics of the language.

The abnormal event hierarchy for μC++ kernel is presented in this chapter. Only dual events are used in the kernel because a single dual event hierarchy is simpler and more flexible. The kernel uses throwing and signalling, indicating that both are significant in the AEHM. Asynchronous abnormal events are found useful in both the kernel and an example. It also eliminates the need of additional built-in features to deal with abnormal conditions encountered in synchronization, implying that the proposed design is indeed comprehensive.

Chapter 5

Conclusion and future work

The focus of this research is about abnormal events as a flow control mechanism in a concurrent programming language. It can be divided into two major parts: first, providing a framework for implementing an AEHM for a concurrent programming language, and second, using μC++ as an experimental platform to implement and apply the framework.

While several concurrent programming languages have abnormal events, rarely does one support both the terminating and resumption models. Those with resumption model, for example, Mesa and Exceptional C, do not properly address the recursive signalling problem. Indeed, many find the resumption facility in Mesa confusing. Consequently, terminating facility is a lot more popular than resumption facility in modern languages.

The design proposed for an AEHM based on the framework is generally applicable and comprehensive. It also eliminates the recursive signalling problem by employing a new propagation mechanism. As a result, the resumption model becomes more attractive and can be introduced to an existing AEHM that currently supports only the terminating model to enhance its functionality.

Furthermore, the framework clearly shows that separating events for the two handling models is not always necessary. Indeed, integrating these events simplifies the AEHM. An abnormal event merely represents a rare condition, and how the system should handle the condition, i.e., choosing the terminating or resumption model, is mostly an independent decision.

An event hierarchy has been shown to be a useful feature in an object-oriented environment. I extend its usage to hierarchical blocking of asynchronous events.

Based on the framework, a comprehensive AEHM for μC++ is implemented. The new features are used to handle abnormal conditions that may arise at runtime. Previously, these conditions are mostly ignored.

In particular, abnormal conditions related to μC++ concurrent programming features including context switching, rendezvous and condition variable are discussed. The language unique orthogonal design requires original ways of using abnormal events. Indeed, the implemented μC++ AEHM retains the overall orthogonal design of the language.

Future work μC++ also provides a set of real-time facilities. Violating a real-time constrain, e.g., missing a deadline, can be treated as an abnormal condition. The new abnormal event features provide a set of tools dealing with these abnormal conditions. However, a study on the abnormal conditions related to the real-time facilities is necessary in order to construct an event hierarchy. The potential use of the throwing and the new signalling mechanism must be investigated.

Appendix A

Glossary

Active execution is an execution bound to a thread. Hence, the thread is carrying out some computation that changes the execution state of the active execution.

Class descriptor describes the different properties of a class being defined. In other words, it specifies if a class has an execution state, if it requires mutual exclusive access, if it has a thread, and if it can be thrown or signalled.

Concurrent environment refers to an environment having multiple threads and multiple executions.

Context switch refers to a change in the binding from a thread to an execution. The execution that gets bound to a thread as a result of a context switch becomes active.

Coroutine environment refers to an environment having one thread and many executions.

Entry is indeed a class method. It is originally used in Ada to refer to a "method" in a task. A task calling an entry may be blocked. The term "entry" is chosen to emphasize the blocking property of a class method in a concurrent OO language.

Execution is an object implementing a sequential computation. Indeed, the object state is the execution state of the sequential computation. An execution can be either a coroutine or a task. A task has a thread, but not a coroutine.

Execution state refers to the context information of a sequential computation including program counter, the stack and global variables.

Faulting execution is the execution propagating and handling a raised abnormal event.

Guarded block is a sequence of statements protected by a set of handlers. These handlers can catch an event propagated into the guarded block.

Handler is a sequence of statements executed when handling an abnormal event. A handler catches a raised event when it is selected the propagation mechanism. The handler is said to have handled the event only if it returns.

Handler clause is the set of handlers associated with the same guarded block. In C++, all the catch phrases of a try block forms the catch clause.

Mandatory agreement refers to the requirement that a signalled event can only be caught by a resuming handler, and a thrown event by a terminating handler.

Non-resumable operation is an operation that is erroneous to continue after an abnormal condition occurs.

Propagating an event in an execution is the step of looking for a handler and directing the control flow of that execution to the chosen handler.

Propagation mechanism defines how to find a handler for a raised abnormal event. Propagating an event into a block means that the propagation mechanism is looking for a handler bound to that block, or more precisely, to one of its invocations.

Raising an event means executing a raise statement statement in this thesis, though it often implies propagating the event and unwinding the stack in other literatures. For asynchronous events, executing a raise statement is clearly separated from propagating the event.

Resume has multiple meanings in this thesis. In the context of coroutine, it is a context switching operation. It is also a handler return mechanism for an abnormal event handling mechanism. Lastly, it refers a thread changing to the running state. The context in which it is used should eliminate any ambiguity.

Signalling is the propagation mechanism without stack unwinding.

Source (execution) is the execution that raises an abnormal event. The source and the faulting execution are different for an asynchronous abnormal event.

Stack unwinding is the destruction of one or more stack frames as a result of a raised abnormal event. Stack unwinding usually happens during the propagation of an event.

Thread is the element capable of executing statements of an execution sequentially. Two or more threads execute statements independently unless they are synchronized. In my concurrency framework, thread and execution state are two orthogonal concepts because a thread can switch from one execution to another. Nonetheless, a "thread" commonly refers to what I call a task, an object having a thread and an execution state.

Throwing is the propagation mechanism with stack unwinding.

Bibliography

[1] Buhr, P. A., and Ditchfield, G. Adding concurrency to a programming language. In *USENIX C++ Technical Conference Proceedings* (Portland, Oregon, U.S.A., Aug. 1992), USENIX Association, pp. 207–224.

[2] Buhr, P. A., Macdonald, H. I., and Zarnke, C. R. Synchronous and asynchronous handling of abnormal events in the μSystem. *Software—Practice and Experience 22*, 9 (Sept. 1992), 735–776.

[3] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G. Modula-3 report (revised). Tech. rep., DEC Systems Research Center, 1989.

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

[5] Castagna, G. Covariance and Contravariance : Conflict without a cause. *ACM Trans. Program. Lang. Syst.* (1995).

[6] Drew, S. J., and Gough, K. J. Exception Handling: Expecting the Unexpected. *Computer Languages 20*, 2 (May 1994).

[7] GEHANI, N. H. Exceptional C or C with Exceptions. *Software—Practice and Experience 22*, 10 (Oct. 1992), 827–848.

[8] GOODENOUGH, J. B. Exception handling: Issues and a proposed notation. *Commun. ACM 18*, 2 (Feb. 1975), 683–696.

[9] GUERBY, L. *"Hypertext Ada 95 Rationale"*. Intermetrics, Inc., 733 Concord Ave., Cambridge, Massachusetts 02138, Jan. 1995. Check http://www.adahome.com/ for an online copy.

[10] INTERNATIONAL BUSINESS MACHINES. *OS and DOS PL/I Reference Manual*, first ed., Sept. 1981. Manual GC26-3977-0.

[11] KENAH, L. J., GOLDENBERG, R. E., AND BATE, S. F. *VAX/VMS Internals and Data Structures Version 4.4*. Digital Press, 1988, ch. 4.

[12] KNUDSEN, J. L. Exception handling — a static approach. *Software—Practice and Experience 14*, 5 (May 1984), 429–449.

[13] KNUDSEN, J. L. Better exception handling in block structured systems. *IEEE Software 4*, 3 (May 1987), 40–49.

[14] KOOENIG, A., AND STROUSTRUP, B. Exception Handling for C++ (revised). In *Proceedings of the USENIX C++ Conference* (1990), USENIX Association, USENIX Association, pp. 149–176.

[15] LISKOV, B. H., AND SNYDER, A. Exception handling in CLU. *IEEE Trans. Softw. Eng. SE-5*, 6 (Nov. 1979), 546–558.

[16] MacLaren, M. D. Exception handling in PL/I. *SIGPLAN Notices 12*, 3 (Mar. 1977), 101–104. Proceedings of an ACM Conference on Language Design for Reliable Software, March 28–30, 1977, Raleigh, North Carolina, U.S.A.

[17] Madsen, O. L., Møller-Pedersen, B., and Nygaard, K. *Object-Oriented Programming in the BETA programming Language.* Addison Wesley, 1993.

[18] Meyer, B. *Object-oriented Software Construction.* Prentice Hall International Series in Computer Science. Prentice-Hall, 1988.

[19] Mitchell, J. G., Maybury, W., and Sweet, R. Mesa language manual. Tech. Rep. CSL–79–3, Xerox Palo Alto Research Center, Apr. 1979.

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

[21] Murer, S., Omohundro, S., Stoutamire, D., and Szyperski, C. Iteration abstraction in sather. *ACM Trans. Program. Lang. Syst. 18*, 1 (Jan. 1996), 1–15.

[22] Stroustrup, B. *The Design and Evolution of C++.* Addison Wesley, 1994.

[23] Tennent, R. D. Language design methods based on semantic principles. *Acta Infomatica 8*, 2 (1977), 97–112. reprinted in [24].

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