Advanced Concepts in Asynchronous Exception Handling

by

Roy Krischer

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2010

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Asynchronous exception handling is a useful and sometimes necessary alternative form of communication among threads. This thesis examines and classifies general concepts related to asynchrony, asynchronous propagation control, and how asynchronous exception handling affects control flow. The work covers four advanced topics affecting asynchronous exception-handling in a multi-threaded environment.

The first topic is concerned with the non-determinism that asynchronous exceptions introduce into a program's control-flow because exceptions can be propagated at virtually any point during execution. The concept of asynchronous propagation control, which restricts the set of exceptions that can be propagated, is examined in depth. Combining it with a restriction of asynchrony that permits propagation of asynchronous exceptions only at certain well-defined (poll) points can re-establish sufficient determinism to verify a program's correctness, but introduces overhead, as well as a delay between the delivery of an asynchronous exception and its propagation. It also disturbs a programmer's intuition about asynchronous propagation in the program, and requires the use of programming idioms to avoid errors.

The second topic demonstrates how a combined model of full and restricted asynchrony can be safely employed, and thus, allow for a more intuitive use of asynchronous propagation control, as well as potentially improve performance.

The third topic focuses on the delay of propagation that is introduced when a thread is blocked, *i.e.*, on concurrency constructs that provide mutual exclusion or synchronization. An approach is presented to transparently unblock threads so propagation of asynchronous termination and resumption exceptions can begin immediately. The approach does not require additional syntax, simplifies certain programming situations, and can improve performance.

The fourth topic explores usability issues affecting the understanding of (asynchronous) exception handling as a language feature. To overcome these issues, tools and language features are presented that help in understanding exception handling code by providing additional run-time information, as well as assist in testing.

For all topics, the necessary extensions to the syntax/semantics of the language are discussed; where applicable, a prototypical implementation is presented, with examples that demonstrate the benefits of the new approaches.

Acknowledgements

Above all, I want to thank my supervisor Peter Buhr, without whose advice and support my work would not have been possible. Further thanks go to the other readers of this thesis, Doug Lea, Lin Tan, Ondřej Lhoták, and Steve MacDonald, for helping me improve it through their comments and suggestions. Finally, I would like to thank Richard Bilson and Ashif Harji, whose advice, which I value dearly, positively impacted my work over many years.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Structured exception-handling introduces a new form of control flow into a programming language. While there are many details and variations, the main contribution of exceptional control-flow is the termination of an arbitrary number of routine invocations and/or the transfer of control to a dynamically-determined location, which are powerful capabilities. In the same sense that exception-handling extends the control-flow features of a sequential programming language, asynchronous exception handling is the analogous extension of the control-flow capabilities of a concurrent language. Instead of treating exception handling and concurrent control-flow as orthogonal concepts, asynchronous exception handling is integrated into and extends the concurrent control-flow.

This thesis covers, at the programming-language level, advanced concepts in asynchronous exception handling. These concepts are *asynchronous propagation control*, *safe full-asynchrony*, *exceptional unblocking*, and *exception usability*, which address questions that arise when asynchronous exceptions are included in a programming language. Any language that includes support for asynchronous exception handling or, in fact, any asynchronous transfer of control (*e.g.*, signals), invariably is confronted with several important questions: How can asynchronous exception handling safely coexist with existing control-flow mechanisms (asynchronous propagation control, safe full-asynchrony)? How can asynchronous exceptions be integrated with existing concurrency constructs (exceptional unblocking)? How can the additional complexities of introducing asynchronous exception handling be tamed so control flow remains comprehensible to the programmer? The answers to these questions, provided by this thesis, constitute building blocks in designing a comprehensive model for asynchronous exception handling in a programming lan-

guage. The following sections define the terms and concepts used in this work, and, in particular, what is meant by 'asynchronous exception handling'.

1.1 Asynchronous Exception Handling

An *exception* is the manifestation of an exceptional situation, which is ancillary to the normal algorithmic path of a program and often considered rare. An exception comes into being when an instance of its *exception type* is *raised*. *Exception handling* is a mechanism by which a raise causes a transfer of control to a block of code called a *handler*. This transfer of control is called *exceptional* or *abnormal* control-flow. *Propagation*, as defined in this thesis, denotes the process by which an appropriate change of control flow as result of an exceptional raise is determined and facilitated, ultimately leading to (but excluding) the execution of a handler matching the exception. A sequential program has a single thread executing on a single stack. Collectively, the stack (or equivalent mechanism), and any other state necessary for execution (*e.g.*, registers, signal masks, *etc.*), make up an *execution*. An exception is said to be *active* if its propagation is in progress; at most one exception can be active in an execution. A complex sequential program (one thread, many stacks), *e.g.*, employing coroutines or continuations, differs from a concurrent program (many threads, many stacks) in that multiple executions can run simultaneously only in the concurrent case. Both complex-sequential and concurrent programs have multiple executions, and thus, can have multiple simultaneously-active exceptions. *Communication* between the raise and its handler is often essential, allowing a handler to receive information from the raise and/or about the location of the raise.

In general, multiple executions do not proceed in isolation; information is communicated to an execution at creation, to/from it during execution, and from it at termination. Normally, communication is performed *synchronously*: One execution stops until the other execution accepts the communication, processes it, and possibly returns a result. A common mechanism for facilitating synchronous communication is a routine call. Within the routine, various forms of control flow are used to delay one execution and restart the other, *e.g.*, suspend/resume for coroutines or wait/signal for tasks. In this complex scenario, a situation, *e.g.*, a failure, in the callee execution can result from or affect the interaction. For example, incorrect data is communicated to the callee, which could raise an exception during processing. However, this situation cannot be remedied in the callee but rather must be rectified by the calling execution. Therefore, the caller has to be made aware of the situation. A natural means to perform the necessary communication is to

transfer the exception between the executions. Hence, mechanisms are necessary for exceptions to cross execution boundaries and for executions to react to these exceptions.

1.1.1 Motivation for Asynchronous Transfer of Control

Inter-execution communication is made more complex when performed *asynchronously* between threads (*e.g.*, out-of-band communication): One execution triggers a change in control flow in another execution to accept a communication. Often an execution must be interrupted and forced to accept a communication, *e.g.*, an exception, at some reasonable point. Unlike the synchronous case, where the exception can only flow from callee to caller, in the asynchronous case, both directions of communication are possible. This kind of communication has its detractors because it is difficult to achieve correctness if transfers occur at non-deterministic points. At the same time, asynchronous communication also seems to be a necessary mechanism in multiple programming domains, *e.g.*, real-time, networks, distributed, control, embedded, *etc.* Possible applications for such a feature are speculative computation, timeouts, user interrupt, or resource exhaustion [MJMR01]. As a result, several programming languages and operating systems attempt to provide at least some rudimentary form of asynchronous interaction (see Section 1.6, p. 13). Exception handling allows for complex and flexible changes in control flow, which makes it suitable for modelling the necessary asynchronous transfer of control. It is therefore important not to think of asynchronous exception handling simply as a way of taking a synchronous exception in one execution and making it accessible to another. While this application is possible, it is too narrow. Instead, this work is concerned with ways for one execution to facilitate an asynchronous transfer of control in another using the exception handling model.

1.1.2 Definitions and Terms

Unlike a regular *synchronous raise* (*e.g.*, throw), which flows into propagation directly, an *asynchronous raise* separates the execution paths of raise and propagation. The *raising execution* performs an asynchronous raise *at* the *propagating execution*, where raising and propagating execution could be the same. The terms *raising task* and *propagating task* are also used. *Asynchronous propagation* is propagation resulting from an asynchronous raise. Similarly, an *asynchronous exception* is an exception raised asynchronously. The propagating execution need not perform any special operations to receive the asynchronous exception, and its propagation can start at arbitrary points (called *full asynchrony*) or only at certain well-defined points (called *re-*

stricted asynchrony), *i.e.*, between any instructions or only at designated points (*poll points*), respectively. The asynchronous nature of the raise results in non-determinism in the propagating execution not present with a synchronous raise. A programming language may provide *asynchronous propagation control* allowing a programmer to indicate which asynchronous exceptions may be propagated; *i.e.*, asynchronous exception types can be enabled or disabled over a body of code denoted by some mechanism in the language. Propagation control can thus mitigate some non-determinism. In detail, asynchronous exception handling involves:

1. *Raise*: a raising execution executes an asynchronous raise statement (analogous to a synchronous raise).
2. *Delivery*: responsibility for the exception is transferred to the propagating execution.
3. *Detection*: a delivered exception is examined and, if eligible according to asynchronous propagation control, its propagation is initiated.
4. *Propagation*: involves handler matching, possible stack unwinding (along the call chain), and transferring control flow to a designated handler matching the active exception, all in the context of the propagating execution.
5. *Handling*: the designated handler, having *caught* the exception, is executed in the context of the propagating execution, deactivating the exception and designating it *handled*.
6. *Completion*: when a handler completes (*i.e.*, it does not *reraise* the handled exception or raise a new one), control transfers to a point after the handler (*termination*) or detection point (*resumption*).

Propagation, handling, and completion are identical to the synchronous case, while detection and delivery are unique to the asynchronous case. Implementations may combine some steps, *e.g.*, raise and delivery, and an asynchronous raise can originate in the run-time system. The sub-system of a language/run-time that is responsible for implementing exception handling is the *exception handling mechanism* (EHM).

Termination involves propagating an exception to a dynamically-located handler, and after completion, control continues at a point lexically after the handler (i.e., static return). This form of exception handling is used when the computation between raise (synchronous case) or detection point (asynchronous case), respectively, and the corresponding handler has to be discarded in light of the exceptional situation, *e.g.*, it contains an unfixable error.

Resumption involves propagating an exception to a dynamically-located handler, and after completion, control returns to the raise point (synchronous case) or detection point (asynchronous case), respectively (i.e., dynamic return). This form of exception handling is used when the computation between raise/detection-point and handler remains correct or can be fixed when the exceptional situation occurs, so execution can continue at the point of detection, *e.g.*, after a correction made by the handler. While the nature of resumption is similar to a routine-call with dynamic lookup, *e.g.*, using function pointers or virtual members, these techniques offer no simple replacement for resumption. In order to look up handlers along the call chain, the equivalent function or object pointers need to be passed down the call chain, which constitutes a maintenance problem: Since all routines whose call the handlers can potentially guard need to reserve a parameter for each handler, their parameter lists can grow to large sizes. Also, whenever a new handler is added in upper-level code, all routines down the call chain need to be augmented by an additional parameter, changing their signatures, which requires recompilation. Hence, such a scheme does not support pre-compiled code. Alternatively, the handling routines can be maintained at a central repository, *e.g.*, a stack of function/object pointers, potentially with additional techniques to allow for the exact guarding of syntactic blocks. Such a scheme basically attempts to re-implement resumption, but is complex and error-prone to maintain by the programmer, and requires the definition of many functions or classes. Language-supported resumption, on the other hand, attaches handlers directly to guarded blocks without burdening the programmer with maintaining the handler stack. It takes advantage of syntactic symmetry with terminating semantics by defining resumption handlers using the same syntax as for terminating handlers, *e.g.*, using a _CatchResume clause instead of a catch clause, making its use more intuitive.

Interruption occurs when the control flow of an execution is altered by something external to the execution. Any asynchronous transfer of control necessarily involves an interruption of the altered execution. A common example of interruption is provided by the (POSIX) *signalling* mechanism [IEE01]. It allows programs to dynamically register routines called *signal handlers* that are called when certain events (*signals*), *e.g.*, timer events, are raised at it by the operating system or another program. On signal propagation, the operating system interrupts the propagating task at a non-deterministic location and transfers its control flow to the appropriate signal handler. If the signal handler returns, the task's control flow resumes immediately after the point of interruption. Hence, signal handling constitutes a form of resumption (see Section 1.6.3, p. 16).

Cancellation is a mechanism through which one execution can terminate another, *i.e.*, the cancelled execution's stack is unwound, and the execution is brought to a terminated state. Cancellation is often performed asynchronously. If an asynchronous exception is handled to completion by a terminating handler immediately preceding an execution's exit-point (*e.g.*, the end of main), the resulting control flow is that of cancellation. Hence, asynchronous exceptions can emulate cancellation, and cancellation is a limited form of terminating exception handling.

1.2 Motivating Example

Putting everything together, a raising execution can raise an asynchronous exception simply by directing it to an identifier associated with the intended propagating execution, *e.g.*,

```
_Throw Ex() _At taskB;
```

The throw does not wait for the exception to be handled or any kind of response, and the raising task blocks for no more than the short time required to facilitate delivery. The task taskB is defined as

```
_Task Worker {
    ...
    void main() {
        ...
        try {
            s0;
          ▷
            s1;
          ▷
            ...
          ▷
            sn−1;
        } catch ( Ex ) { ... }
    }
} taskB;
```

and propagation (▷) can theoretically occur between any two statements s_i or even between arbitrary instructions. For taskB to properly handle Ex, it must be ensured that asynchronous propagation is confined to within the try-block, which is the role of asynchronous propagation control (see Chapter 2).

For a concrete scenario in which asynchronous exceptions can be useful, imagine tasks involved in a conversation [BW97], *e.g.*, two tasks using sensor data to determine aircraft position. Each task queries its own sensor, which can take some time; if one fails, the others' data becomes irrelevant. Hence, if a task fails, it needs to inform the other task involved in this conversation, so they can terminate the conversation together. An asynchronous exception with terminating semantics is well-suited for facilitating this control flow. Its propagation terminates the sensor-

querying code and its handler can restart the failed conversation, *e.g.*,

```
for ( int attempts = 1; ; attempts++ )
    try {
        try {
            try {
                res = read_sensor();                    // read sensor data
            } catch ( ... ) {
                reset_sensor();                         // cleanup in case of any failure
                _Throw;                                 // reraise
            }
        } catch ( MySensorReadFailure msrf ) {
            SensorReadFailure fail( msrf );             // document reason for overall failure
            _Throw fail _At otherTask;                  // inform the other task
            _Throw fail;                                // throw synchronously as well
        }
        if ( all_success() )
            break;                                      // if all succeed, break out of loop
    } catch ( SensorReadFailure srf ) {                 // otherwise
        /* analyze srf to know what happened */
        if ( attempts == MAX_TRY )                      // if maximum retries reached
            _Throw;                                     //   escalate issue, otherwise retry
    }
```

If one of these tasks cannot fulfill its part, it needs to inform the other, which is done using a SensorReadFailure exception. The handler for SensorReadFailure decides whether to retry reading the sensor, or escalate the problem to higher-level code by reraising the exception. Note that cancellation is not a suitable replacement for asynchronous exception handling in this case as local state (the attempts count) needs to be preserved to correctly complete the protocol. Other local state, hidden in the example, might exist if this particular sensor reading is but a small part of the overall task, *e.g.*, taking multiple readings and calculating an average. In fact, the position-reading conversation could be part of a larger trajectory-determining conversation, which could itself be part of a larger automated-landing conversation. In general, even when no local state must be preserved, the cost of cancelling and disposing of an execution with subsequent creation of a replacement can be computationally more expensive than re-using the existing execution after a handled exception.

Resumption can be employed when actions are not to be terminated, but rather additional information or a correction is supplied. In the preceding example, imagine one sensor-reading task discovers a high variance in its readings and decides to take additional readings. This fact can be communicated to the other task by an asynchronous exception using resumption semantics. The propagating task executes the following code:

```
for ( int i = 0; i < max; i++ )
    try {
        ...
        res = read_sensor();                        // read sensor data
        ...
    } _CatchResume( TakeMoreReadings tmr ) {        // in case of resumption propagation,
        max = tmr.newmax;                           //   update max and return to point
    }                                               //   of interruption
```

If it receives a TakeMoreReadings exception, the propagating task interrupts its work, executes the resumption handler (which updates max), and then, after the handler completes, resumes its work at the point of interruption.

1.3 Identifying Potential Exception Occurrences

All code locations in which exceptions can be propagated should be known to the programmer. Otherwise, an exception may go unhandled and terminate the program, or even if the exception is handled, propagation can start inside *vulnerable code*, *i.e.*, code in which propagation leaves the program in an inconsistent state.

□ More precisely, the locations where exceptional control-flow can occur should be known. Note, in this work, the term *propagation* and its derivatives are used universally as the cause for exceptional control-flow inside the propagating execution. It is possible to distinguish between the initial raise of an exception and its propagation up the call chain since, in the synchronous-raise case, exceptional control-flow can be due to a raise, or due to propagation caused by a call to a routine that raises/propagates an exception. However, it is possible to think of a synchronous raise as a propagation through a hypothetical throw routine[1], so all exceptional control-flow in the synchronous case can be regarded as due to propagation. In the asynchronous-raise case, a possible exceptional control-flow can only occur in the propagating execution, and thus, due to propagation. The term 'propagation' can therefore be used without loss of generality in all cases of exceptional control-flow. □

Using local information about which exceptions can propagate, it is possible to verify that no exception propagations can occur inside vulnerable code, or that all vulnerable code is protected

[1]There are C++ compilers that convert throw statements into routine calls, *e.g.*, to __cxa_throw.

(*e.g.*, by handling exceptions, or using the resource acquisition is initialization (RAII) technique [Str97]). It is further possible to leverage such local information and extend guarantees to higher levels, *e.g.*, the STL vector<T> :: erase method guarantees not to raise an exception if T's copy constructor/assignment cannot raise one [Abr98]. For sequential programs, identifying the potential propagations can be as simple as code inspection and informal annotation of called routines. However, this approach relies on external documentation (*e.g.*, for pre-compiled code) and is error-prone. More formal methods exist in the form of *exception specifications*, also called *exception lists* [BHM02, §6.5]. Languages with *single-level propagation* (an exception can only be handled by the immediate caller), such as in Goodenough's work [Goo75] or CLU [LS79], naturally require the declaration of raisable exceptions for each routine as the raisable exceptions are an integral part of the interface between caller and callee. This approach provides a way to identify all potential propagations, but can be tedious for the programmer and over-clutter the code. In languages that allow multi-level propagation, *e.g.*, Java, Ada, C++, the handler site can be far from the raise site, yet every routine call along the path between raise and catch propagates the exception, with the potential to leave the program in an inconsistent state. C++ and Java provide a means to explicitly declare the exceptions that can be propagated by a routine. While there are arguments against the use of exception specifications [BHM02, Eck07], they help to ensure code safety and verifiability of a program when used properly. At compile-time, the Java compiler checks the agreement of throws-declarations with the rest of the program, *i.e.*, either the corresponding exceptions are handled within the caller of the routine, or the caller declares them in its own exception specification (it propagates them). Exceptions involved in this static process are called *checked* exceptions. C++'s check occurs only at run-time, and unless explicitly declared otherwise, any exception can propagate through a routine, which significantly reduces the usefulness of this feature. However, it can sometimes aid in more efficient code generation and does not burden the programmer with the tedium of maintaining throw declarations.

Java also has *unchecked* exceptions [GJSB00], which are exceptions whose potential raise need not be declared by a routine, and hence, are not part of the static check. The reasons for declaring an exception as unchecked are that it can be raised in a great multitude of locations (*e.g.*, arithmetic overflow), and explicit declaration would be too tiresome for the programmer, or that they logically cannot be anticipated by the program. The latter case is due to a programming error detected by the run-time system (*e.g.*, using an illegal subscript to access an array) or an entirely external cause, *e.g.*, a hardware failure. Unchecked exceptions can cause problems since

they open a hole in the safety of the exception specification checks, potentially subverting the underlying mechanism[2]. For this reason, it is recommended to only use unchecked exceptions for unrecoverable errors, *i.e.*, errors that are not caught [ZHR+], reducing the propagation of an unchecked exception to merely a named unwinding of the stack.

1.4 Interruptibility Problem

Asynchronous exceptions are similar to unchecked exceptions in that they can originate from outside the propagating execution (though not necessarily from outside the program), or may not be a direct consequence of the propagating execution's actions. More importantly, since they are asynchronous, the starting location of their propagation inside the propagating execution cannot be anticipated. This non-determinism can cause serious errors in a program if asynchronous exceptions start propagating within vulnerable code (*e.g.*, recall the STL guarantee about not raising exceptions inside certain methods). Buhr *et al.* call this the *non-reentrant problem* [BHM02]. However, this name is misleading as there need not be a re-entrance of some routine for the problem to occur. If propagation starts when program data is in an inconsistent state and consistency cannot be restored before this data is accessed, the program fails. For example, imagine a list manipulation routine being interrupted by an exceptional propagation while the list is in an inconsistent state, and termination semantics cause the rest of the routine to be aborted; alternatively, imagine a resumption handler accessing that list as part of recovery. While re-entrance is certainly a case in which this problem can occur, a better term to describe this phenomenon in general is the *interruptibility problem*.

Recall from Section 1.3 the two ways to ensure exception safety:

1. Verification that no exception propagation can occur within vulnerable code: Unlike the synchronous case, asynchronous propagation need not originate from a routine call, but can occur anywhere. Hence, if a region in which asynchronous exceptions can occur contains vulnerable code, this vulnerable code needs to be protected.

2. Protection of vulnerable code through handlers or cleanup constructs (*e.g.*, finally in Java[3] or

[2] Incidentally, Eckel promotes a technique that converts checked into unchecked exceptions, so that the mechanism can be subverted more efficiently [Eck07].

[3] Weimer [Wei06] points out the complexities of using finally correctly, and presents the idea of the *compensation stack*, a language extension to ensure the fulfillment of commitments such as resource cleanup in the presence of exceptions.

RAII in C++).

Protection of vulnerable code is significantly more difficult in the asynchronous case. Consider the example in Figure 1.1. Resources db1 and db2 need to be released/closed, but only if they have been acquired/opened. The left example attempts to solve the problem by handling the exception. In the synchronous case, assuming only the call to potentialRaise can propagate exceptions, it suffices to close db1 and db2 in the catch clause (and re-propagate the exception) to protect this code from exceptions. In the asynchronous exception case, assuming that asynchronous exceptions can be raised anywhere within this code, it is impossible to tell where exactly propagation begins. It could begin between the calls to db1.open and db2.open, in which case closing db2 inside the handler could be an error. In fact, the exception could occur before or during db1.open, so closing db1 in the handler may not be correct either.

The example on the right attempts to solve the problem using the RAII approach by closing db1 and db2 inside the destructor ~RAIIopen, which is only executed if its corresponding constructor RAIIopen, which contains the open, runs to completion. However, suppose the asynchronous exception starts propagation after the call to db.open but before the constructor completes. Since the Database object is open, it needs to be closed, but since construction of the RAIIopen object is incomplete, its destructor is not executed, and the Database object is not closed. If the programmer tries to improve this solution by writing

```
RAIIopen() try {                    // special constructor-try-block
    DB.open();
} catch (...) {
    DB.close();
    throw;
}
```

then it can still fail if the exception propagates while inside the try-block, but before any actions are committed that require the closing of DB, *e.g.*, while executing the first statement of Database::open. The call to DB.open itself could be wrapped into an RAII-wrapper, as well as the individual statements that comprise open, *etc.*, and such a scheme might possibly work. In general, however, this is not a usable programming method.

The interruptibility problem can be characterized as follows. In the synchronous case, a contiguous piece of code contains a few well-defined locations at which exceptions can propagate and interrupt the contiguous control-flow. Safety can be maintained by protecting these few locations with the methods presented above, which produces a kind of atomicity with regard to synchronous exceptions that is suitable for the usual programming methodology of structuring

try/catch

```
...
Database db1, db2;
try {
    db1.open();
    db2.open();
    potentialRaise();
    ...
    db1.close();
    db2.close();
} catch (...) {
    db1.close();
    db2.close();        // necessary?
    throw;              // reraise
}
```

RAII

```
struct RAIIopen {
    Database &DB;
    RAIIopen(Database &db)
      : DB(db) { DB.open(); }
    ~RAIIopen() { DB.close(); }
};
Database db1, db2;
{
    RAIIopen raiidb1( db1 );
    RAIIopen raiidb2( db2 );
    potentialRaise();
    ...
}
```

Figure 1.1: Protecting vulnerable code from exceptional propagation

programs out of smaller building blocks. The non-determinism of asynchronous exceptions conceptually breaks up contiguous pieces of code into instruction-sized pieces. Apart from the fact that programming languages generally do not offer the granularity to protect single instructions[4], it is difficult and error-prone to protect this many locations in order to form one exception-atomic region of code. Add to this the possibility of pre-compiled code (whose source is inaccessible) that is also subjected to asynchronous exceptions.

1.5 Asynchrony Models

As stated in Section 1.1.2, p. 3, there are different models of asynchrony. *Full asynchrony* has the greatest non-determinism where propagation can potentially begin at any instruction. This kind of asynchrony matches the common intuition and is assumed implicitly in the previous sections. Examples of full asynchrony include POSIX threads (pthreads) cancellation in its asynchronous mode, or POSIX signals [IEE01].

A different model is *restricted asynchrony*[5], in which control-flow is still non-deterministic, but the propagation (or interruption) causing this non-determinism is restricted to occur only at certain well-defined locations called poll points. Poll points are like routine calls that, in the case of exceptions, can propagate a synchronous exception, and thus, the programming methodology used in the synchronous case to deal with exceptions can also be used with restricted asynchrony. Upon encountering a poll point, the list of delivered (pending) exceptions is traversed, and if one

[4] See [Cha94] for suggestions on how to implement asynchronous exceptions by breaking a program down into small (potentially instruction-sized) pieces and providing restartable handlers protecting these pieces.

[5] [MJMR01] calls it *semi-asynchrony*.

is enabled, it is propagated. Examples of restricted asynchrony include pthread cancellation in its deferred mode (POSIX calls poll points *cancellation points*), or μC++ up to version 5.6.0. *Pure poll points*, *i.e.*, routines whose only purpose is to poll, are a convenient way for a programmer to trigger propagation. Examples include pthread_testcancel and μC++'s uEHM::poll.

The motivation behind restricted asynchrony is that programming under full asynchrony is very difficult. By restricting asynchrony to a few well-defined points in a program, verifying its correctness becomes much easier compared to the full-asynchrony model. In addition, a common programming mind-set is to implement the main algorithm first, and then, often (and unfortunately) as an afterthought, deal with deviations such as errors or exceptions (*i.e.*, boundary cases) [SGH10]. Restricted asynchrony appeals to this mind-set as the programmer starts out with normal deterministic code as a base line in which a few abnormalities (propagations through poll points) need to be dealt with subsequently. Full asynchrony, on the other hand, implies that nondeterminism is the base line, and a programmer has to struggle with the interruptibility problem, *i.e.*, reason about deviating (exceptional) control-flow, essentially at all points in order to write correct programs. This approach is more complex and also contrary to the common programming mind-set, so it is not surprising that it is more error-prone than using restricted asynchrony.

On the other hand, restricted asynchrony can cause delays when timely propagation is important since control needs to reach a poll point for propagation to occur. With full asynchrony, no delay is necessary as propagation can begin immediately. This trade-off is similar to the general problem in concurrency, where serializing constructs make it easier to write predictable programs, but at the same time restrict concurrency, and thus, performance. Furthermore, restricted asynchrony requires the programmer to be aware of what operations are poll points, and where these are located in a program. This knowledge may not always be intuitive, and the need to have poll points for propagation may require unintuitive programming idioms. Finally, polling incurs a run-time cost even when no exceptions are being raised.

1.6 Related Work

A number of publications address extending exception semantics to a multi-execution domain, and they are classified in three different ways.

1.6.1 High-Level Systems

These systems are largely concerned with asynchronous exceptions only as a means to implement high-level mechanisms, *e.g.*, distributed interactions, and do not examine the underlying language mechanics, or the problems inherent in asynchronous transfer of control, *e.g.*, [CR86, RXR98b, MT02, PRP04].

1.6.2 Synchronous Propagation

Other work mentions asynchronous exceptions, but not in the sense as they are used in this work. An asynchronous computation of some kind causes exceptions to cross from one execution to another, but this transfer requires an explicit action by the propagating execution, *e.g.*, a message receive [IY91], a synchronization with the raising execution [Iss91], or the access of a future-value [Rin06]. Other examples include [KO02, CC05]. Since exception propagation is synchronous (in some form) in the propagating execution, the issue of asynchronous transfer of control is not addressed, which is at the heart of this thesis.

1.6.3 Asynchronous Propagation

Some publications, and a few main-stream languages, address an asynchronous exception model similar to the one used in this thesis, *i.e.*, one that is primarily characterized by an asynchronous transfer of control. Systems supporting only restricted asynchrony technically can be classified as employing synchronous propagation. Whether their propagation is considered asynchronous shall additionally depend on a subjective assessment of how often poll points occur, how surprising the occurrence is, and whether there are other language mechanisms that can cause surprising exceptional control-flow generated by asynchronous interaction with another execution.

Szalas and Szczepanska [SS85] propose an hypothetical concurrent language in which an asynchronous signal (exception) is caused by a raise to another process (execution). The system supports termination and resumption semantics, as well as asynchronous propagation control, and works under full asynchrony. It does not permit synchronization between processes, nor are raised exceptions retained when their target is inactive (blocked) or has exception propagation disabled.

In [FFS96] Fleiner *et al.* discuss the problems of performing thread cancellation in an object-oriented system such as *pSather*. They conclude that based on their criteria, "[...]it is not possible

to have an easy, safe and efficient parallel, object oriented language [...] that offers a way to asynchronously stop a thread." In this thesis, I am going to show that, based on my criteria, intuitive, safe asynchronous exception handling (and thus cancellation) is indeed possible with acceptable performance.

An extension to Concurrent Haskell [MJMR01] employs an asynchronous-exception model similar to the one in this work, albeit in a functional language. Exceptions are explicitly raised at another thread using the throwTo primitive. It supports limited propagation control using block and unblock, and operates under full asynchrony. Asynchronous propagation while holding a mutual exclusion lock is implicitly disabled, which can postpone propagation indefinitely (*e.g.*, see [FFS96]). Asynchronous exceptions may be propagated under certain circumstances even if propagation is disabled.

Erlang's message sending is asynchronous, but its message reception is synchronous, *i.e.*, requires an explicit receive [Eri]. Two processes can be *linked*. A terminating process emits an *exit signal* to all processes to which it is linked; these exit signals are received asynchronously and, if termination was abnormal, cause the abnormal termination of the recipient. A process can *trap* exit signals, which turns them into messages, thus deferring the 'propagation' of a received exit signal, and enabling its handling. This trap mechanism can be considered a limited form of propagation control. Processes can emit exit signals without terminating, and a *kill* message cannot be trapped, *i.e.*, cancellation cannot be controlled by the cancellee.

Modula-3's Alert mechanism [Bir89] provides thread interruption functionality. A thread can alert another, which turns on the target's alert-pending status. If an alerted thread calls TestAlert (or AlertWait), it propagates an Alerted exception.

Ada [Int95] allows an exception to cross execution boundaries during rendezvous synchronization. It also offers an *asynchronous transfer of control* (ATC) facility through asynchronous select on the client-side [BW97, §10.7];[BW03], which crudely approximates asynchronous exceptions.

Java's Thread class supports a stop method, which facilitates thread cancellation, but can be used to raise arbitrary Throwable objects inside the called thread. An exception raised in this way between two executions (threads) is fully asynchronous. However, catching an exception raised through the stop mechanism is only permitted if it is reraised subsequently, *i.e.*, the handler must not complete; thus, the mechanism is only suitable for thread cancellation as opposed

to general asynchronous exception handling. Furthermore, Thread.stop is deprecated in current Java versions for safety reasons [Sun]. Similarly to Modula-3, Java supports thread interruption through Thread.interrupt, which can interrupt a thread under restricted asynchrony (requiring an interrupted thread to explicitly check its status) or, using the real-time extension, full asynchrony [BHR02]. This mechanism cannot directly raise an arbitrary exception, nor can threads be asynchronously interrupted (using real-time semantics) while executing synchronized statements, further limiting this feature's use: For example, similar to the limitations of Concurrent Haskell, threads holding a resource through a synchronized statement cannot be cancelled through asynchronous exceptions.

The .NET framework supports asynchronous thread cancellation through the Abort mechanism of System.Threading [Mica], which is similar to Java's Thread.stop; it can only raise ThreadAbortException.

POSIX threads [But97, IEE01] also support thread cancellation, which can either be *deferred* (restricted asynchrony), with checking at cancellation points in a limited number of system routines, or (fully) *asynchronous*. POSIX signals cause an asynchronous call to a dynamically-determined routine, and thus, constitute a crude form of asynchronous resumption, but are a heavy-weight feature requiring user/kernel mode switching. POSIX is an operating-system interface, not a programming language.

Note, thread cancellation does not constitute exception handling because the cancellation cannot be handled, and thus, only provides a very limited form of termination. Java, Concurrent Haskell, Ada, and .NET[6] do not support resumption. Modula-3, Java and .NET do not support certain high-level concurrency-concepts, *e.g.*, Ada-style rendezvous[7].

Finally, [BM00] proposes an asynchronous-exception model that serves as the basis of this thesis. It is characterized by explicit asynchronous raises, restricted asynchrony, asynchronous propagation control, and support for termination semantics using the C++ model, as well as resumption using proprietary syntax. Its shortcomings are the restricted asynchrony it employs, as well as the inability of blocked tasks to propagate asynchronous exceptions. Part of this thesis is concerned with mitigating these shortcomings, as well as their analogues in other languages.

[6] Visual Basic supports a form of synchronous resumption.

[7] This restriction is relevant in Chapter 4.

1.7 About This Work

To avoid misunderstandings or misconceptions about the nature of this work, the following sections explain its focus and level of abstraction.

1.7.1 What This Thesis is About

Throughout this thesis, it is implicitly assumed that asynchronous exceptions are a useful feature that is best integrated directly into a concurrent programming language (see Section 1.1.1, p. 3). Source code, being the expressions of a programming language, is therefore found throughout this document. μC++ is the main language of study for which the features proposed in this work are implemented; therefore, μC++ is naturally the language of choice for many of these source code examples. Appendix A contains a short introduction to μC++'s extensions to C++, and gives an overview of the features discussed in this thesis. The model of asynchronous exceptions used in this work is μC++'s [BM00]. Since μC++ is based on C++, and C++-like models are often assumed in this thesis, understanding of C++, particularly its exception-handling model and mechanism, is essential for understanding this work (see [KS90, Str94, Str97]). Nevertheless, the majority of the concepts presented in this work are directly applicable to most modern programming languages, and many of its ideas and implementations are transferable to other systems with similar notions of asynchronous exception handling or transfer of control. For ease of understanding, some of the example programs in the main part of this thesis are incomplete or use simplified syntax; the complete versions with full syntax can be found in Appendix B.

1.7.2 What This Thesis is Not About

This thesis deliberately does not give an introduction into the development of structured exception-handling, its motivation and advantages, major milestones, or its proliferation and prevalence. There are many interesting publications that cover these areas, *e.g.*, [Goo75, LS79, Cri82, Knu84, YB85, Geh92, LS98, Kri02, BHM02, RS03].

While some of the motivation for the existence of asynchronous exceptions is touched upon in Section 1.1.1, p. 3, an exhaustive examination of such a motivation constitutes a thesis in itself, and is therefore not the focus of this work. Readers who still dispute the usefulness of asynchronous exceptions are encouraged to suspend their disbelief and evaluate this work on its own merit given the assumptions from Section 1.7.1. This work does not define a new model

for the transfer of asynchronous exceptions from one execution to another. It builds upon the model devised in [BM00], thoroughly analyzes parts of this model, and extends the model with advanced features. This thesis does not analyze the merits of different asynchronous-exception models as such analysis is beyond its scope. Readers who do not agree with the asynchronous exception model used in this thesis are encouraged to evaluate this work on the basis of how it improves upon the underlying asynchronous exception-handling model.

This thesis is not about software engineering or distributed systems. Obviously, programming languages are the tools employed in software engineering, and the results in this thesis can be employed when implementing a distributed system, but such additional work occurs at a higher abstraction level and is not within the scope of this thesis. In particular, this work does not try to interpret how multiple agents can collaboratively handle an exceptional situation, what an exception in one thread might appear like to another thread, or how real-world exceptional situations are best modelled as (asynchronous) exceptions[8]. A limited selection of interesting publications that do focus on such topics are [CR86, RXR98a, MT02, DUV06, FFM+10, KT10].

1.7.3 Remainder of This Document

Having established the asynchronous-exception model assumed herein and the issues surrounding the interruptibility problem, the rest of this thesis addresses several areas around this topic. Chapter 2 thoroughly explores the concept of asynchronous propagation control, which has an essential role in writing programs with asynchronous exception handling. Building upon the analysis and classification of that chapter, Chapter 3 then proposes a new approach of adjusting the asynchrony model depending on the propagation control that is in effect, in order to allow for safe, more intuitive programming with potentially better performance. Chapter 4 explores the issue of raising an asynchronous exception at a task that is blocked, and shows how to resolve this issue for a variety of blocking instruments. Finally, Chapter 5 explores how the additional complexities of programming with asynchronous exceptions can be tamed by providing powerful ways to test asynchronous-exception code, as well as additional run-time information to assist programmers' understanding.

[8] There are no examples featuring travel agents in this document, but there is one featuring a barber.

Chapter 2

Asynchronous Propagation Control

Even with restricted asynchrony, programming with asynchronous exceptions is still difficult, and the non-determinism introduced by the interruptibility problem can affect a program in unexpected ways. As mentioned in section Section 1.4, p. 10, determinism can be restored by designating regions of code as uninterruptible or exception-atomic. To deal with the non-determinism caused by concurrency, atomicity is generally achieved by employing locks. However, locking does not help with the interruptibility problem since only one thread is involved in propagation. Hence, exception-atomicity requires a mechanism designed specifically for this purpose, *i.e.*, to restrict the set of asynchronous exceptions that can be propagated in an execution within a region of code. I refer to this mechanism as *asynchronous propagation control* (*e.g.*, see [BMZ92, p. 766];[BM00, §16.3]), which this chapter explores in detail. It shows the essentiality of asynchronous propagation control, analyzes different approaches including combining approaches, and discusses the implications each approach has with respect to asynchronous-propagation semantics. This chapter serves as a reference for language designers wishing to support asynchronous exception handling.

The essential nature of asynchronous propagation control with full asynchrony is easy to demonstrate. Consider the following example:

```
unsafe_to_interrupt();          // unsafe to interrupt
try {
    safe_to_interrupt();        // safe to interrupt with respect to Ex
} catch ( Ex ) { }
```

Suppose the call to safe_to_interrupt can be safely interrupted by a propagation of Ex, with the try-block and catch-clause placed to handle the exception. However, this example is still incorrect

as there is no way to guarantee that a propagation of Ex cannot start outside the try-block, *e.g.*, in unsafe_to_interrupt. Hence, there must be a way to restrict propagation to a region in which it can be properly handled, the try-block in this example, which is the role of asynchronous propagation control.

With restricted asynchrony, propagation is restricted to occur only from poll points. Situations such as above can therefore be avoided if no poll points are encountered outside of the try-block. However, poll points may be frequently encountered in code that cannot be interrupted safely or that cannot handle all possible exception propagations, *e.g.*, inside the call to unsafe_to_interrupt. While one solution is to make such code safely interruptible, such a conversion is generally difficult and may be impossible depending on the circumstances. Hence, there must be a way for a program to protect itself against propagations occurring at such poll points.

To further illustrate the need for asynchronous propagation control, consider Java's Thread.stop mechanism. It allows for fully-asynchronous thread cancellation, but has no asynchronous propagation control. Consequently, Thread.stop has been found to be unsafe in general and has been deprecated in more recent Java versions [GJSB05, Sun]. Similarly, the lack of propagation control for thread abortion in Microsoft's .NET framework leads to function calls like Monitor.Enter gaining keyword-like semantics as they cause code-generation restructuring in order to ensure correct cleanups [Duf07].

The ability to turn all propagation off or on is the minimum mechanism required (*e.g.*, [MJMR01]). However, a region of code may be able to safely handle the propagation of one exception type but not the propagation of another. Therefore, it is preferable to specify a set of exception types whose propagation is allowed or disallowed. Furthermore, with full asynchrony, the initial state, *i.e.*, before the first explicit use of propagation control, should be such that no asynchronous propagation should be allowed; otherwise an execution could be interrupted before it has an opportunity to turn off asynchronous propagation. For symmetry and ease of use, the same semantics should apply for restricted asynchrony as well.

With asynchronous propagation control, if asynchronous propagation of a set of exceptions is allowed, the set (or its propagation) is said to be *enabled*, otherwise, it is *disabled*. There are several different approaches to asynchronous propagation control, each with different scope or extent. Scope refers to the visibility of the state introduced by a propagation-control directive, *i.e.*, is the state visible only in the local scope or everywhere? Similarly, extent refers to the life

time of such state, *i.e.*, does it have a limited life-time with a definite beginning and ending at compile-time, or does it persist indefinitely?

2.1 Dynamic

The simplest version is called the *dynamic approach* and is characterized by an infinite scope and extent, *i.e.*, the propagation-control state is visible in every scope and persists until changed explicitly. Asynchronous propagation is dynamically enabled until it is disabled by an opposing call. Such a call pair encloses a region of asynchrony, *i.e.*, one that is interruptible with regard to the asynchronous propagation of a set of exceptions, *e.g.*,

```
enableException( Ex );    // enable propagation of Ex
...                       // can be interrupted with regard to Ex
safe_to_interrupt();      // can be interrupted with regard to Ex
...                       // can be interrupted with regard to Ex
disableException( Ex );   // disable propagation of Ex
```

whereas the opposite pairing is used to create a region of atomicity with regard to the asynchronous propagation of a set of exceptions, *e.g.*,

```
disableException( Ex );   // disable propagation of Ex
...                       // propagation of Ex precluded
unsafe_to_interrupt();    // (unsafe) propagation of Ex precluded
...                       // propagation of Ex precluded
enableException( Ex );    // enable propagation of Ex
```

The extension to Concurrent Haskell [MJMR01] has unblock and block operations to enable and disable, respectively, the propagation of all exceptions using dynamic semantics. Its propagation control does not distinguish between different exception types, *i.e.*, it only works on all exception types. A possible cause for this restriction is that exception types in Haskell cannot be easily extended and organized in a hierarchy, and thus, one exception type is often used universally[1]. Another example of the dynamic approach are the enableContext/removeContext routines in the *Guardian* model [MT02].

Unix/POSIX signals also work under the dynamic approach, being controlled by calls to sigprocmask [IEE01]. POSIX threads [But97], too, employ a dynamic approach to control cancellability, *i.e.*, the pthread_setcancelstate and pthread_setcanceltype routines. A pthread's cancellation cannot commence after the most recent call to pthread_setcancelstate is performed with the argument PTHREAD_CANCEL_DISABLE. In this way, a pthread can protect sections of its code from cancellation. Pthread cancellation is enabled as the default, which is a reasonable

[1] Marlow proposes a solution for this limitation and its integration with OOHaskell [Mar06].

design decision as cancellation is rare and terminates the thread (*i.e.*, cannot be handled).

In theory, using paired calls of pthread_setcancelstate with PTHREAD_CANCEL_DISABLE and PTHREAD_CANCEL_ENABLE, it is possible to protect the code between the calls from interruption through cancellation and structure the code thus.

The simple dynamic approach allows almost arbitrary structuring of asynchronous propagation control, but the infinite extent implies a need to pair dynamic calls precisely, which constitutes a potential source of error. If a programmer forgets to disable propagation after enabling it, the program may be interrupted in unexpected places. This situation is analogous to the use of mutual-exclusion locks in concurrency; while providing a flexible mechanism to ensure mutual exclusion, the need to precisely pair acquisition and release of a lock is a potential source of error.

2.2 Semi-dynamic

In order to avoid the pairing issues mentioned above, the *semi-dynamic approach* to asynchronous propagation control tries to add robustness to the dynamic approach while sacrificing flexibility. It is characterized by an infinite scope and a finite extent, *i.e.*, the propagation-control state is visible in all dynamically nested scopes and persists until the end of a block. Concurrent Haskell's unblock and block operations are also available as scoped combinators, *e.g.*, block (vulnerableCall x), which have semi-dynamic characteristics.

Another example of the semi-dynamic approach is employed by μC++. The _Enable / _Disable statement enables or disables the asynchronous propagation of exception sets within its dynamic scope. Consider the example in Figure 2.1. In this example, all code inside the _Enable block and outside of the _Disable block is logically designated as interruptible (by DeadlineAlarm and ServerFailure exceptions). In particular, statements s1 and s2, as well as the calls and execution of doSomething and doSomethingElse can be interrupted by arriving DeadlineAlarm and ServerFailure exceptions (and all exceptions derived from these). The call to doNotInterrupt cannot be interrupted by any asynchronous exception, and propagation is deferred at least until the end of the _Disable block (assuming doNotInterrupt does not contain _Enable statements). Note that all asynchronous exceptions are initially disabled in μC++ to guarantee that try-blocks with handlers can be set-up before an exception can propagate.

An unusual example of propagation-control can be found in [SS85], which employs enable and disable statements to control propagation of an arbitrary set of exceptions. The effects of

```
void foo() {
    _Enable < DeadlineAlarm > < ServerFailure > {
        s1;
        doSomething();
        s2;
        _Disable {              // means disable all exceptions
            doNotInterrupt();
        }
        doSomethingElse();
    }
}
```

Figure 2.1: Controlling asynchronous propagation in μC++

these statements are visible inside the *module* in which they are issued and in all modules called from it, *i.e.*, infinite scope, but revert once the module is left. Classifying the extent of this approach depends on what exactly is meant by module. If module refers to a procedure, then the extent is finite. If module refers to a translation unit (*e.g.*., Pascal unit), then the extent, while technically still finite, would be so large as to be almost infinite–a three-quarter-dynamic approach, possibly. In this system, an exception raised, but whose propagation is disabled, is lost. These semantics do not seem appropriate for an asynchronous setting. Since the raising and the propagating execution are not synchronized, the source cannot anticipate whether the target is ready to propagate the exception, with the result that the exception may be lost. While the raising execution is notified whether its exception is propagated, it is difficult to imagine writing predictable programs under this scheme without requiring some synchronization between raising and propagating execution.

2.3 Static

When both extent and scope are finite, *i.e.*, the propagation-control state persists until the end of a block and is only visible when statically therein, I call the resulting approach *static*. Imagine a hypothetical Static_Enable statement, which results from taking a semi-dynamic _Enable statement and restricting its scope to its lexical block, *e.g.*, in

```
void bar() {
    ...                         // E is not enabled when called from foo
}
void foo() {
    s1;
    Static_Enable < E > {
        s2;                     // E is enabled because inside lexical scope
        bar();                  // call leaves lexical scope
        s3;                     // E is enabled because inside lexical scope
    }
    s4;                         // E is not enabled because beyond scope/extent
}
```

propagation of E is enabled for statements s1 and s2, but not while control is inside the body of bar.

A real-world example using this approach is the real-time specification for Java (RTSJ) [BBD+00], which specifies that exceptions of type AsynchronouslyInterruptedException can be asynchronously propagated within routines whose checked-exception specification includes this exception type. Java exception-specification semantics apply statically within a routine; the resulting asynchronous propagation control can therefore be classified as having finite scope and extent (static approach). For example, in the following RTSJ code

```
class ServerFailure extends AsynchronouslyInterruptedException { ... }
class SynchronousEx extends Exception { ... }
...
public void foo() throws ServerFailure, SynchronousEx {
    s1;
    someCall();              // Finite scope: ServerFailure not enabled inside call
    s2;
}
```

the statements s1 and s2 are interruptible[2] by ServerFailure, whereas execution of someCall is not (unless that routine has asynchronous exceptions in its throws-list). Exception SynchronousEx can only be propagated synchronously.

Note, it is unclear whether these asynchronous propagation semantics were chosen and then the exception-specification mechanism adapted because it provided the desired scope/extent, or if conversely, the static propagation-control is simply a consequence of attaching propagation-control semantics to Java's exception specifications. In either case, it seems unsatisfactory to restrict propagation-control granularity to entire routine bodies since exception handling itself (*i.e.*, placement of try-blocks and handlers) has finer granularity. Furthermore, the concept of exception specifications is orthogonal to propagation control (also see Section 2.6.2, p. 49), so combining these two is questionable. In particular, when using such combined exception-specification/propagation-control semantics, exceptions that can be raised asynchronously can only be (re-)raised synchronously if the exception is caught within the raising routine; otherwise, if the exception is propagated to the caller, it must be listed in the routine's exception specification, which implicitly turns on asynchronous propagation where it is not necessarily safe. For example, imagine in the previous example that foo is called by another function bar. Recall that

[2]While non-call poll-points are conceivable, poll points are usually function calls, and calling such a function moves control out of the scope of static propagation-control. For simplicity of the discussion, it is therefore helpful to assume a full-asynchrony approach with static propagation-control.

foo can be asynchronously interrupted by ServerFailure:

```
public void bar() throws ServerFailure {
    s1;
    try {
        foo();
    } catch ( ServerFailure e ) {
        ...                          // analyze problem
        throw e;                     // cannot fix problem, reraise exception to caller
    }
    s2;
}
```

The asynchronous ServerFailure exception is propagated synchronously into bar where it is potentially reraised (synchronously) to its caller. Since it can potentially propagate a ServerFailure, Java requires bar to list it in its exception specification. However, due to its double-duty as propagation control, asynchronous propagation of ServerFailure is now enabled within bar itself–a subtle and potentially critical error as there is no guarantee that s1 and s2 are safely interruptible. Even if a programmer realizes the potential problem, rectifying it, *e.g.*, by modularizing s1 and s2 into routines in order to move them out of the scope of the exception-specification/propagation-control, requires significant code restructuring. Finally, RTSJ's propagation is always disabled in synchronized blocks, *i.e.*, those that provide mutual exclusion, which is a restriction of the language not dictated by the concept of propagation control itself.

2.4 Semi-static

When the scope is finite and extent is infinite, *i.e.*, the propagation-control state introduced is visible inside a limited scope, *e.g.*, a block, and is remembered when control re-enters the scope, I call the resulting approach *semi-static*. An equivalent concept with the same attributes is a local C++ variable with static storage class, which is visible only inside the scope in which it is defined (finite scope), and persists for the life time of the program (infinite extent).

With such a combination, it is unclear what the correct scope bounds need to be in order to gain additional semantics compared to the static approach. If the scope is defined in a strict sense, *i.e.*, as for the declaration of variables, the resulting semantics are that the propagation-control directive is visible from the point of its issuance until the end of the current block, *e.g.*,

```
void bar1() {
    ...
    if ( condition ) {
        s1;
        enable E;                    // hypothetical semi-static directive
        s2;
    } else {
        ...
    }
    s3;
}
```

In the example above, statement s1 does not see that E is enabled, and only s2 sees the enabling as the scope is exited after s2. However, the semantics of this example are equivalent to those of the following example using static propagation control:

```
void bar2() {
    ...
    if ( condition ) {
        s1;
        Static_Enable < E > {      // hypothetical static directive
            s2;
        }
    } else {
        ...
    }
    s3;
}
```

It is easy to see that with such strict scoping rules, the semi-static approach provides no additional semantics compared to the static approach. Note, the RTSJ approach from Section 2.3, p. 23 could conceivably be classified as semi-static since the semantics of the static and semi-static approach are identical using RTSJ's routine scope. However, it seems prudent to classify it under the simplest scheme that allows for the observed semantics, which is the static approach.

Extending the definition of scope for the semi-static approach produces unintuitive results. For example, a routine-scope could be defined, implying changes made to propagation control inside the routine revert once the routine terminates, but are restored when the routine is called again and are visible in its entirety. Such semantics would be surprising for many programmers. Reducing the scope to just the block in which the directive resides does not help. If, in the example above, bar1 is called and condition is true and then called again with a true condition, E would be enabled in s1 even though the enabling statement occurs further down in the block. When condition is false when bar1 is called initially, then when statement s1 is first encountered, propagation of E is disabled. If the scope is defined in some arbitrary manner, *e.g.*, from the point of the enable/disable until the routine scope is left, the result is equally confusing, *e.g.*, what is the propagation-control state at s3? While unambiguous semantics can be defined, the behaviour

is going to be unusual, and it seems questionable whether anyone could intuitively write correct programs using such schemes.

In conclusion, the semi-static approach is either redundant or difficult to understand, and is therefore ignored for the remainder of this document. Table 2.1 summarizes the different classifications for asynchronous propagation control based on scope and extent.

2.5 Finite *vs.* Infinite Extent

The existence of propagation-control implementations with finite and infinite extent justifies a discussion about the merits of each approach. Since finite extent always implies some kind of block being employed, this approach shall also be denoted *block-based* and these two terms are used interchangeably where appropriate. Similarly, infinite extent implies the use of some kind of routine call, so the term *routine-based* shall have the same meaning.

While this section studies extent in the context of asynchronous propagation control, the discussion can largely apply to any block/begin-end language construct in comparison to its routine-based/single-statement analogue. For example, an exception handler guarding a try-block has block characteristics, and a mechanism that registers handler routines, *e.g.*, by pushing them onto a central cleanup-handler stack, is its routine-based counterpart. A similar analogy exists between monitors and mutex locks [Bri73].

To illustrate finite extent, μC++'s _Enable / _Disable blocks are used. For infinite extent, a hypothetical routine

```
prop_control_t  set_prop_control( exception-type, prop_control_t )
```

is used, which is modelled after pthread's cleanup-control functions or those controlling POSIX signals. The set_prop_control routine has two parameters, the first of which specifies the exception type controlled, and the second whether it is to be disabled or enabled. It also returns the state of propagation control before the routine is invoked (for easy restoration). Restricted asynchrony (poll) is assumed for simplicity in this section.

2.5.1 Advantages of Finite Extent

Consider the example in Figure 2.2. The block structure of _Disable guarantees that while inside the lexical scope of the block, the propagation of E exceptions is disabled. Hence, a programmer is always aware of the exceptions that can be propagated inside a block; this concept shall be

Scope \ Extent	∞	$\not\infty$
∞	dynamic (*e.g.*, pthread_setcancelstate)	semi-dynamic (*e.g.*, μC++ _Enable)
$\not\infty$	semi-static	static (*e.g.*, RTSJ)

Table 2.1: Propagation-control approaches

```
_Disable < E > {
    bar();
    poll();          // cannot propagate E
}
```

Figure 2.2: An example of propagation control using finite extent

called *block awareness*. Block awareness is a by-product of syntactic requirements with regard to proper block nesting: In order to use blocks correctly, a programmer needs to be aware of where it starts, where it ends, and what statements it contains. Violation of block-nesting rules results in a compile-time error, which is the preferred time at which errors should manifest themselves. The most important advantage from a practical standpoint is that in using a propagation-control block, a programmer cannot forget (without causing a compile-time error) to close a block and thus turn off the propagation-control directive that the block establishes. The role of finite-extent propagation-control compared to the infinite-extent approach is thus analogous to the way monitors ensure mutual exclusion in a more structured way, automatically releasing resources and preventing deadlocks due to forgetfulness, compared to mutual-exclusion locks.

Note, block awareness does not extend into the dynamic scope (see discussion in Section 2.6, p. 48). For example, inside of bar, E exceptions may be enabled, and subsequently propagated up the call-stack and into the _Disable block. Nevertheless, it is certain that all code inside the lexical scope of the _Disable block, *e.g.*, a call to poll, cannot trigger an E exception to be propagated. This guarantee removes uncertainty and helps the programmer in statically analyzing the program, leading to more robust code.

2.5.2 Disadvantages of Finite Extent

While this method is intuitive and very robust, it precludes certain forms of control. In particular, it may be necessary to enable the propagation of an exception inside a block, but with the actual propagation occurring outside that block, *e.g.*,

```
void turniton() {
    // enable E}
void useit() {
    poll();
}

_Disable < E > {
    turniton();
    useit();
}
```

In the above example, the programmer needs the poll inside useit to propagate an E exception. If enabling E is done through the block-based approach (*i.e.* _Enable <E> { } in turniton), then clearly, this enable has no influence on code inside of useit, and subsequently, the call to poll from useit cannot propagate an E exception. Only if the enabling of E persists beyond block boundaries – through some mechanism that has a 'global' effect, *e.g.*, a function call like set_prop_control(E, ENABLE), can procedure turniton ensure that the call to poll inside useit causes propagation of an E exception.

Practical Restrictions

The situation above can occur with software libraries, which may sometimes require the programmer to follow a certain sequence of function calls (protocol) that communicate with each other or their invoker using asynchronous exceptions, *e.g.*, opening and closing an I/O device.

More specifically, consider a display_ok_cancel_box routine that one thread calls in order to create another thread to display and manage a dialog box. The user's choice is then communicated back to the original thread through an asynchronous resumption exception (say BoxResult). In order for this scheme to work, the original thread needs to have the propagation of BoxResult enabled. Ideally, display_ok_cancel_box should enable the propagation of the exception as this enabling is necessary exactly when that routine is used, *e.g.*[3],

```
void * ok_cancel_box( void *arg ) {
    int res = GUI_dialog_box( CANCEL );        // display dialog box and wait for result
    _Resume BoxResult( res ) _At arg;          // forward result to original thread
}                                              //   through asynchronous raise
void display_ok_cancel_box() {
    set_prop_control( BoxResult, ENABLE );     // routine-based enable of BoxResult
    startThread( ok_cancel_box, thisTask );    // start new thread, does NOT block
}
```

[3]This example uses a hypothetical language inspired by pthread-like threading and μC++-like exception handling.

```
void eventLoop() {
    ...
    try {
        ...
        display_ok_cancel_box();                    // display box without blocking
        for ( ... ) {
            ...                                     // continue independent work
        }
    } _CatchResume( BoxResult e ) {                 // resumption handler
        set_prop_control( BoxResult, DISABLE );     // routine-based disable of BoxResult
        /* extract result and use it */
    }
}
```

With block-based propagation-control, such a design is impossible since enabling inside display_ok_cancel_box only affects code called/executed from within that routine, but not code that gets executed after display_ok_cancel_box returns. Instead, the programmer must wrap the entire code sequence in which the exception could be propagated into an _Enable block, *e.g.*,

```
void eventLoop() {
    ...
    try {
        ...
        _Enable < BoxResult > {
            display_ok_cancel_box();                // display box without blocking
            for ( ... ) {
                ...                                 // continue independent work
            }
        }
    } _CatchResume( BoxResult e ) {                 // resumption handler
        /* extract result and use it */
    }
}
```

Note that in this example, unlike in the previous ideal solution, BoxResult *remains* enabled after returning from the resumption handler, *e.g.*, inside the for-loop, which may be undesirable. Furthermore, imagine display_ok_cancel_box is called from within another (much larger) library routine. The programmer must enable propagation for the call of the large library routine as a whole. Finally, if, unlike above, the resumption handler for the BoxResult exception is provided by the library as well, the programmer requires a deeper understanding of implementation details in order to place the _Enable block correctly for an exception it never handles.

A similar problem occurs when user code is executed by library routines through a call-back mechanism as there is no easy way for a user to wrap the affected routine invocations with an appropriate _Enable / _Disable block. Imagine an open and a close routine provided by the programmer but called by library code. Without control over the invoking (library) code, the use of block-based control to enable an exception for exactly the time between the calls to open and close is impossible. For example, ideally, a programmer may want to write

```
void open_socket() {
    /* open communication socket */
    set_prop_control( ServerFailure, ENABLE );
}
void close_socket() {
    set_prop_control( ServerFailure, DISABLE );
    /* close communication socket */
}
int main() {
    ...
    generic_IO_send( open, close, ... );          // invoke library routine, supply call-backs
}
```

but with only block-based propagation-control, the programmer must write

```
int main() {
    ...
    _Enable < ServerFailure > {
        generic_IO_send( open, close, ... );      // invoke library routine, supply call-backs
    }
}
```

instead, which enables the propagation of ServerFailure over the entire invocation of the generic_IO_send library routine.

Syntactic Restrictions

The finite extent approach suffers from another class of restrictions resulting from the nature of a language block. Imagine a parameter being passed to a routine that determines whether a certain exception should be propagated, *e.g.*,

```
void foo( bool propE ) {
    if (propE)
        set_prop_control( E, ENABLE );
    else
        set_prop_control( E, DISABLE );
    /* algorithmic code */
    ...
    poll();
}
```

With block-based propagation-control, the same scenario requires substantial code duplication or restructuring:

```
void foo( bool propE ) {
    if ( propE )
        _Enable < E > {
            /* algorithmic code */
            ...
            poll();
        }
    else
        _Disable < E > {
            /* algorithmic code */
            ...
            poll();
        }
}
```

In an attempt to avoid code duplication, it may be possible to implement the block-based approach to accept dynamically evaluated directives, *i.e.*, _Mask <true, E> instead of _Enable <E> and _Mask <false, E> instead of _Disable <E>. With such a block-based mechanism, the code from above can be written as

```
void foo( bool propE ) {
    _Mask< propE , E > {
        /* algorithmic code */
        ...
        poll();
    }
}
```

Alternatively, the algorithmic code can be factored into a routine and the call to this routine duplicated instead of the entire code, reducing the problem. This kind of solution may still require extensive rewrites, *e.g.*, when variables are used inside the algorithmic code, but these variables are defined outside of it, or if the algorithmic code contains return or goto statements.

It is possible to conceive of other situations in which the required block structure and its implications (*e.g.*, with regard to the life time of objects) conflict with the intended program design. After all, a propagation-control block is merely supposed to indicate a range of instructions in which certain exceptions can be propagated, whereas a syntactic block has more extensive semantics. Consider an object which is instantiated automatically inside a propagation-control block. The propagation-control semantics do not require its life time to be restricted to within that block, but the rules for syntactic blocks do.

Note that, in general, it is impossible to separate the propagation-control block from the block structure of the language (and thus avoid the previously described interaction) as this can produce situations with unclear semantics. Imagine the following piece of code:

```
void foo() {
    if ( propE ) {
        _Enable < E > {
    }//if
        }//_Enable
}
```

Such a construction has no intuitive semantics and conflicts with the rules of control flow. Hence, even if propagation-control blocks were not fully-featured language blocks, they would at least have to be properly contained withing the block structure of the program (nesting).

Finally, since block activation occurs on the run-time stack, it can only affect the current execution. Imagine a scenario in which one execution wants to send an exception to another execution, even if the recipient is not prepared to handle it. In order to ensure exception propa-

gation, the raising execution would have to enable propagation for the propagating execution. A slight extension to the routine-based approach, *i.e.*, an additional parameter to specify the target execution in set_prop_control could be used in this case. Note that such a situation differs from the propagation-control concepts introduced so far, and would probably only occur in trusted situations among tasks, but nonetheless, the block-based approach cannot be used in this case.

2.5.3 Advantages of Infinite Extent

A propagation-control approach with infinite extent is more general than the block-based one as

```
_Disable < E > {
    ...
}
```

can be emulated[4] by

```
{
 prop_control_t oldSetting = set_prop_control( E, DISABLE );

    ...
 set_prop_control( E, oldSetting );
}
```

and, unlike block-based propagation-control, this approach allows enabling/disabling an exception between arbitrary blocks.

2.5.4 Disadvantages of Infinite Extent

A propagation-control approach with infinite extent also has several disadvantages.

Error-proneness

Since routine-based control affects the entire program/execution, regardless of where they are issued, the routine-based approach is more error-prone and can lead to counter-intuitive results as in the transformation of the example from Figure 2.2, p. 28:

```
prop_control_t oldSetting = set_prop_control(E, DISABLE);
bar();
poll();              // may or may not propagate E
bar();
set_prop_control(E, oldSetting);
```

It is uncertain what can happen when poll is called since the propagation-control state may be changed inside bar and not reset, *e.g.*

[4] A slightly more complex emulation using RAII is needed to ensure exception-safety.

```
void bar() {
    ...
    set_prop_control(E, ENABLE);
}
```

The more powerful ability to affect propagation control beyond block boundaries inevitably introduces higher complexity and greater potential for errors.

Vulnerable Handlers

As a further disadvantage, infinite extent can make it difficult to safely handle an exception since the type of the propagated exception is still enabled when control transfers to the handler [MJMR01], *e.g.*, in

```
try {
    set_prop_control(E, ENABLE);    // enable E
    poll();                         // propagate E exceptions
} catch ( E ) {
    /* E is still enabled here */
    set_prop_control(E, DISABLE);   // too late?
    ...
}
```

when control reaches the handler, there can be a race between the disabling of E and potential additional propagations of E, even if the call to set_prop_control(E, DISABLE) is the first statement inside the handler[5]. POSIX signals circumvent this problem by implicitly disabling further signals of the same kind inside the signal handler (by default). A possible solution to the problem above is for the catch handler to have analogous semantics, *i.e.*, E is automatically disabled within the handler. Alternatively, if mixing of block- and routine-based propagation-control is possible, an explicit disabling can be achieved by writing

```
catch ( E ) _Disable < E > {        // E is explicitly disabled for the extent of the handler
    ...
}
```

where the semantics of such a catch-specific _Disable guarantee that potential propagation can only occur inside the block, *i.e.*, when the disable directive is in effect (also see the accept-specific try-block in Section 4.3.2, p. 102).

Summary

In most cases, the functionality provided by the block-based propagation-control approach is sufficient and even preferred, so replacing it entirely by the routine-based method is undesirable

[5] Imagine full asynchrony in this case.

given its disadvantages. Nonetheless, as Section 2.5.2, p. 28 illustrates, there are situations in which the block-based approach is lacking and for which the availability of the routine-based approach is useful. Hence, it could be advantageous to support both approaches in one EHM.

2.5.5 Block and Routine Conflicts

Supporting two such different approaches for propagation control is problematic as there may be situations where one conflicts with the other, and for these situations, there must exist a clear strategy to avoid or resolve the conflict. There are two basic scenarios in which a conflict occurs:

Routine conflict: A routine-based control directive conflicts with an earlier block-based one, *e.g.*,

```
_Disable < E > {
    ...
    set_prop_control(E, ENABLE);
    ...
}
```

Block conflict: A block-based control directive conflicts with an earlier routine-based one, *e.g.*,

```
set_prop_control(E, DISABLE);
...
_Enable < E > {
    ...
}
```

Note, a conflict does not have to be between an enable and disable, it can equally occur between an enable and enable or a disable and disable (*homogeneous conflict*) where one is routine-based, and the other block-based. Also, there may be even more cases/conflicts to distinguish if enabling and disabling are assigned different priorities, *e.g.*, disables are regarded as more important and are thus made stronger, *i.e.*, able to supersede a conflicting enable. However, as the following sections establish, homogeneous conflicts are irrelevant, and enable and disable are analogous and of equal priority. Hence, without loss of generality, only the conflicts in which an enable follows a disable in the manner depicted above are investigated here.

Homogeneous Conflicts

With regard to homogeneous conflicts, *i.e.*, between enable and enable or disable and disable, *e.g.*,

```
_Enable < E > {
    ...
    set_prop_control( E, ENABLE );
}
```

it is clear that actual conflicts can only arise with regard to an earlier opposite directive, a disabling of E in the example above. So the conflict reduces to how the _Enable < E > behaves with regard to a preceding set_prop_control(E, DISABLE) (block conflict), or how the set_prop_control(E, ENABLE) behaves with regard to a surrounding _Disable < E > (routine conflict). Similarly, in a homogeneous conflict like

```
set_prop_control( E, ENABLE );
...
_Enable < E > {
    ...
}
```

there is no conflict between the two propagation-control directives depicted, but rather a potential conflict with regard to a preceding directive. As the following section shows, enable and disable are completely analogous, so no further distinction is necessary.

Prioritizing

A further distinction could be made if a conflict between an enable following an earlier disable were resolved differently from one in which a disable follows an earlier enable (independent of whether they are block-based or routine-based). This distinction is motivated by the fact that enable and disable mainly serve different purposes in a program even though they are analogous in principle.

Indeed, the most common case in which a user wants to disable asynchronous propagation occurs when a section of code needs to execute without interruption due to stack unwinding from propagation, *e.g.*,

```
/* suppose propagation of E is enabled here */
_Disable {                      // needs to be at least as strong as preceding enable
    /* protected code */
}
```

Note, again, this protection is only effective for the extent of the block, but can be sufficient if the programmer has adequate knowledge of the code therein.

Clearly, the situation above requires making disable at least as strong as enable. Hence, in the block conflict case, disabling E works as intended as the _Enable is at least as strong as the preceding disable. Block awareness inside the _Enable block is therefore preserved. In the routine conflict case, in order to maintain block awareness, disabling needs to be even stronger than enabling in order to still protect the rest of the lexical block from incoming/pending exceptions; *i.e.*, the routine-based enabling of E would have no effect as it is overruled by the earlier disable.

Note, the initial setting would usually be (and is for μC++) for the propagation of all exceptions to be turned off as this allows for guarded regions to be set-up safely. Now, if disabling had higher priority, it would be impossible or at least difficult to enable any exceptions at all since any subsequent enabling (using either form of propagation control) would be defined to be weaker. The obvious solution would be to treat the initial implicit disable differently from an explicit disable; this, however, results in yet another increase in design complexity, along with the non-trivial question of how to restore this initial state after it has been changed.

On a more abstract level, it can be argued that enabling an exception is just as critical for correct control-flow as disabling. Imagine a task waiting for an exception to be propagated in order to continue execution, *e.g.*,[6]

```
_Disable {                          // disable propagation of all exceptions
    _Enable < E > {                 // now just propagation of E is possible
        for ( ;; ) {
            poll();                 // busy wait for E
        }
    }
}
```

In this case, the task polls actively until an E exception gets propagated, after which the stack is unwound and the for-loop terminated so that the task can continue its execution after the handler (not depicted). If the _Disable directive (in its block- or routine-based form) had higher priority than the _Enable statement, the loop would continue forever.

Note that in this case, the E exception is used as a way of synchronizing between tasks. While such active polling is clumsy at best, it is equally conceivable that a task could be signalled/awakened from a condition variable by means of an exception, *e.g.*,

```
_Disable {
    _Enable < E > {
        cond.wait();
    }
}
```

This intuitive method of signalling (see Chapter 4) would be precluded by giving disabling higher priority than enabling.

Theoretically, it may be possible to distinguish between empty _Disable statements (*i.e.*, those that disable all exceptions) and those that block specific exception types; by making the empty _Disable block at most as strong as the enabling one, the above example would work even if a 'regular' disable had higher priority than enable. However, this solution would only work in this particular case, and in general, the resulting semantics would be very complex and confusing.

[6]Note that the block-based syntax in this example is chosen merely for convenience.

Therefore, disable should not have higher priority than enable.

In conclusion, since disable should not be stronger than enable and enable should not be stronger than disable, clearly, both have to be treated equally. So, without loss of generality, it suffices to only examine conflicts in which an enable follows a disable, *i.e.*, there are only the two basic cases, routine and block conflict as defined in Section 2.5.5, p. 35, to consider.

2.5.6 Conflict Resolution

It has already been established that the simple block-based approach suffices in many cases, but not all. Furthermore, due to the block structure and its resulting block awareness, this method is easier and more intuitive to use correctly. As a result, logical mistakes are rare, and some can even be detected at compile-time (*e.g.*, wrong block nesting). Hence, supporting block-based propagation-control in an EHM is desirable. In order to also support the remaining cases in which block-based control cannot be used, routine-based functionality needs to be present as well. However, having these two approaches coexist in one EHM requires a strategy to resolve the resulting routine and block conflicts. The following sections present different options for conflict resolution.

Block-based > Routine-Based ("BSR")

In order to take advantage of block awareness, routine-based propagation-control cannot be as strong as its block-based counterpart. The result of such a design decision is that for the block conflict, E is disabled until it gets enabled for the entire block, and upon exiting the block, it is disabled again. For the routine conflict example, the situation is more complex: As the routine-based enabling of E cannot override the surrounding _Disable block, the routine-based directive has to be either

1. ignored,
2. rejected,
3. or deferred.

The first option means that no state change occurs when encountering the routine-based directive. The second option generates some form of error, *e.g.*, aborting the program. Both of these cases are difficult to use since a programmer may not always be aware of the surrounding block structure of a call to set_prop_control (*e.g.*, inside a library routine, and assuming infinite scope). The

first case is difficult to recognize and debug, while the second case makes it difficult to write a program that correctly anticipates all possible occurrences of the error.

In the third case, the enabling of E is deferred until all conflicting surrounding blocks are exited. Only then is propagation of E exceptions enabled in a routine-based fashion. Note, this option still allows for all exceptions to be turned off initially in order to set up guarded regions. However, when one propagation-control method has priority over the other (as in this case), the initial disabling directive must be of the weaker kind, or otherwise it is impossible to enable propagation with the weaker method. This initializing requirement can be achieved by using an implicit routine-based disable in the BSR case. Note that this situation is somewhat analogous to the introduction of a 'weak' initial disable when distinguishing priorities between disable and enable.

Routine-Based > Block-Based ("RSB")

In the opposite scenario, in which routine-based propagation-control has precedence over block-based one, the block conflict is resolved by ignoring or rejecting the block-based _Enable directive. Note that deferring is impossible in this case as the routine-based directives effectively always determine the current state. For the routine conflict, assuming E is not enabled with the routine-based method beforehand, the exception is disabled for the part of the block before the call to set_prop_control, and enabled afterwards, which disturbs block awareness, *e.g.*, in

```
void bar() {
    ...
    set_prop_control(E, ENABLE);
}
void foo() {
    _Disable < E > {
        s1;
        bar();
        s2;
    }
}
```

propagation of E is disabled up to the call to bar (actually, up to the call to set_prop_control inside bar), but enabled after the call, *e.g.*, in s2, despite s2's being located inside of a _Disable block.

Note that when making routine-based propagation-control stronger than block-based, in order to be able to make use of the block-based approach at all, a third state (in addition to enabled and disabled) has to be introduced to routine-based propagation-control, *e.g.*, *unset*, with rules on how

to restore this state (*e.g.*, enable and disable cancel each other out), *e.g.*,

```
/* state of E is unset */
set_prop_control(E, ENABLE);
/* state of E is now enabled */
set_prop_control(E, DISABLE);
/* state of E is unset again */
_Enable < E > {
    /* state of E is enabled because previous state is unset */
}
```

The semantics of unset need to be like a weak disable, *i.e.*, a disable that can be overridden by any other block- or routine-based directive. Otherwise, without an unset third state, as soon as a routine-based directive is encountered, it overrides all future block-based directives because it has infinite extent (and scope), *e.g.*,

```
set_prop_control(E, ENABLE);
/* state of E is now enabled */
set_prop_control(E, DISABLE);
/* state of E is now disabled */
_Enable < E > {
    /* state of E is disabled because the routine-based disable overrides this */
}
```

In addition, analogously to BSR, the initial implicit disable directive must be of the weaker kind, in this case block-based, or unset. Note that the addition of a third state for the stronger (block-based) method is unnecessary in BSR since an unset state implicitly exists outside of an _Enable/_Disable block. One could argue that such an unset, *i.e.*, a weak disable, state exists initially, before any propagation control is encountered.

Block-based == Routine-Based ("BER")

A third alternative is to give both approaches equal priority and let run-time precedence resolve conflicts, *i.e.*, the later directive overrides the earlier. This approach means a routine conflict is resolved as in the previous RSB case and the block conflict like in BSR. Note that theoretically, it is possible to introduce a third, unset, state for this case as well. However, this addition complicates the design and makes programs harder to understand with no obvious advantage.

2.5.7 Discussion

Clearly, of all cases, RSB is the least desirable for an EHM with two propagation-control methods since it barely takes advantage of the block-based approach. This imbalance can be mitigated by the introduction of an unset state, complicating the overall design. Using the routine-based approach exclusively is simpler and still effective as it allows for maximum flexibility by the

programmer, but at the cost of the challenges in writing correct programs mentioned before.

BER has the advantage of being very simple with clear semantics. This simplicity, however, comes at the cost of losing block awareness. After each routine call inside a _Enable / _Disable block, the current propagation-control state could have changed with the unfortunate consequence that what the programmer sees (the lexical block) is the opposite of the actual state. This disadvantage may be alleviated by introducing a routine to query the current propagation-control state, *e.g.*, state = query_prop_control(E). However, it is clear that such a routine only gives dynamic insight–it cannot yield the static clarity that block awareness achieves.

BSR is the only case which preserves block awareness. As previously mentioned, this awareness makes the programmer's task easier and less error-prone. On the other hand, routine-based directives are somewhat disadvantaged as they only affect propagation control outside of a _Enable / _Disable block dealing with the same exception type.

Effect of BSR on Libraries

A further issue with BSR arises from the fact that program behaviour as a result of a call to set_prop_control depends on the context in which this call occurs. For example, if a library routine uses routine-based propagation control, and the code calling it has surrounding propagation-control blocks with respect to the same exception, then its call of set_prop_control may not produce the desired result due to BSR's resolution strategy for routine conflicts. This problem especially affects library routines as they usually have no way of knowing the context from which they are called. The responsibility therefore falls upon the programmer of the library to document its use of routine-based propagation-control, while the user of such library routines needs to make sure to not use block-based propagation-control of the same exception higher up in the call stack. Such documentation can be done verbosely, which requires the programmer to follow a strict documentation convention describing what kind of routine-based propagation-control is used in a routine. Alternatively, such a convention can be enforced syntactically, similarly to exception specifications, *e.g.*,

```
void display_ok_cancel_box() controls( BoxResult ) {
    ...
}
```

In this way, the compiler can help the programmer (using the library routine) locate possible routine conflicts.

Coding Conventions

With BSR, it is still effective to use block-based propagation-control of the same exception *lower* in the call chain than a routine-based directive. Thus, libraries that use block-based control exclusively implicitly avoid conflicts, *e.g.*,

```
void library_routine() {
    _Enable < E > {                    // guaranteed to work under BSR
        ...
    }
}
void user_code() {
    ...
    set_prop_state( E, DISABLE )       // safe to assume this disable works if
                                       //   all code that calls user_code is known
    library_routine();                 // call library routine whose implementation is unknown
  /* after return, safe to assume that E is disabled (again)
     here if libraries only use block-based propagation-control */
}
```

This asymmetry between routine and block conflict can be seen both as an advantage and a disadvantage: While it allows for one way of safely mixing block-based and routine-based propagation-control, it also creates an asymmetry since libraries cannot take advantage of routine-based propagation-control.

It should be noted that block-based propagation-control is sufficient in most cases, and its use is encouraged in all cases in which it is effective. Furthermore, if a library needs to use routine-based control, it is unlikely that this approach affects exception types for which the caller has provided blocks for specific exceptions. In this case, a routine conflict that results in an 'unusual' behaviour of set_prop_control (*i.e.*, it is ignored, rejected, or deferred) should occur rarely. However, in the case of unspecific blocks that control propagation of *any* exception, *e.g.*, _Enable { ... }, a routine conflict is more likely. This problem is exacerbated by the convenience of such blanket propagation-control, which suggests it is used frequently. A solution for this problem is to provide a way to exclude certain exceptions from the blanket propagation-control blocks, *e.g.*, if BoxResult is an exception type used in a routine-based fashion by a library, a user of that library who is aware of the potential routine conflict can then write

```
_Disable < ! BoxResult > { ... }
```

which is understood as disabling any exception but BoxResult. While such a feature can help avoid routine conflicts, it also increases the overall design-complexity.

If deferring is chosen as a way of conflict resolution, BSR can create counter-intuitive results since routine-based directives do not take effect immediately. A possible work-around is to wrap

the routine-based directive inside a block-based directive, *e.g.*,

```
_Disable {
    ...
    _Enable < E > {                         // enable E immediately
        ...
        set_prop_control(E, ENABLE);        // make sure E remains enabled
    }                                       //   outside the _Disable block
    ...
}
/* E implicitly enabled here */
```

In the above example, the _Enable block ensures that propagation of E is enabled immediately, while the routine-based directive aims to ensure that it remains enabled in the future. Note that the routine-based propagation-control only takes effect outside the _Disable block.

Querying the Propagation-Control State

As previously mentioned, it may appear advantageous to determine the current propagation-control state. There are two possible applications for such a query: to conditionally poll for exceptions, and to avoid conflicts. With conditional polling, it may be desirable to poll under the condition that an exception E cannot be propagated consequentially. Such a conditional poll can be achieved by

```
if ( query_prop_control( E ) == DISABLE )
    poll();
```

This situation can occur if a piece of code is incapable of handling the propagation of E exceptions. However, if E exceptions cannot be handled by a piece of code, their propagation can be more easily prevented by explicit disabling while still allowing other exceptions, *i.e.*,

```
_Disable <E> {
    poll();
}
```

It is conceivable that a (library) routine provides different implementations depending on whether propagation of an exception is enabled, so the availability of query_prop_control is still useful (also see Section 5.3.2, p. 141).

In the second possible application of avoiding conflicts, it is conceivable that a (library) routine could query the propagation-control state and then decide what method, *i.e.*, block- or routine-based, to use in order to avoid conflicts with upper-level propagation-control, *e.g.*,

```
bool routinepropc = false;
if ( query_method( E ) == ROUTINE ) {
    set_prop_control(E, ENABLE);
    routinepropc = true;
} else {
    _Enable <E> {
        ...
    }
}
...
if ( routinepropc == true )
    set_prop_control(E, DISABLE);
```

It is clear that such an approach may require substantial code duplication or restructuring and increases the overall complexity. Note also how when set_prop_control is used, an opposite call to set_prop_control is (usually) required so that the program must remember what method of propagation-control was chosen. Furthermore, this querying capability is only useful for RSB. In the cases of BSR and BER, it suffices for the (library) routine to use block-based propagation-control in order to avoid conflicts. If the use of routine-based propagation-control is required for correct functionality, then there is no way to avoid a potential conflict and querying the propagation-control state cannot help in this case. Considering the problems with RSB and the increase in code size and complexity that is required in any case, the ability to query the propagation-control method is hardly useful.

2.5.8 Resolution *vs.* Avoidance

Both BSR and RSB avoid conflicts rather than actually resolving them: Since one method of propagation-control overpowers the other, conflicts are either avoided by transforming the program into one without conflicts (deferring or ignoring) or the program generates an error when it encounters a conflict.

In contrast, BER actually has to resolve conflicts since both methods have equal priority. As a consequence, the semantics of block-based propagation-control have to be evaluated in more detail. In particular, it is difficult to decide how to restore the previous state upon exiting a block. There are two basic approaches: change removal and restoration. In order to distinguish the two, the following definitions are necessary:

Let (E, i) be the propagation-control state with regard to exception E before a directive concerning the propagation of E is issued (*i.e.*, an _Enable/_Disable block is entered or set_prop_control is called). Then, when the directive is issued, the state is replaced by $(\mathsf{E}, i+1)$.

Change Removal

Change removal means that when an _Enable / _Disable block is exited, only those changes to the propagation control it implemented on entry are restored. More precisely, it means that when such a block is exited, the previously saved state (E, i) is restored *only* if the current state is $(\mathsf{E}, i+1)$. The actual propagation-control state (*i.e.*, enable or disable) is irrelevant, only the identity (expressed through *i*) of the state is important. In other words, with change removal, any directives established at the beginning of the block, should they still be in effect, are taken out at the end of the block. Consider the following example:

```
set_prop_control( E, ENABLE )        // state becomes (E, i)
_Enable <E> {                        // state becomes (E, i+1)
    set_prop_control( E, DISABLE )   // state becomes (E, i+2)
} // exit from block
```

Here, when the second call to set_prop_control returns, the propagation of E is disabled. Since the _Enable block is not responsible for creating this state (the current state is $(\mathsf{E}, i+2)$), the propagation-control state remains unchanged when exiting the block, and E remains disabled. This approach has the advantage that routine-based changes persist beyond block boundaries, which is consistent with the idea behind routine-based propagation-control. The disadvantage is that the routine-based approach is favoured slightly and counter-intuitive situations as above are still possible, namely, in which the block directive is _Enable < E >, yet the propagation of the exception is disabled because a call to set_prop_control(E, DISABLE) occurs within the block. Note, this is a situation in which BER violates block awareness.

Restoration

Restoration means that when exiting a propagation-control block, the previously saved state (E, i) is restored unconditionally, *i.e.*, after leaving the block which set the state to $(\mathsf{E}, i+1)$, the propagation-control state with regard to E becomes (E, i). In the above example, upon exiting the block, the old state for E is restored to (E, i), enabled. This change occurs regardless of the routine-based disabling of E that occurs within the block.

The restoration approach has appeal because it is compatible with the RAII programming idiom [Str97]. In fact, if the routine-based approach is used exclusively in an EHM and the block-based approach emulated as in Section 2.5.3, p. 33 with RAII or finally blocks, the resulting scheme has the semantics of BER with restoration.

However, it is problematic that restoration favours the block-based approach by allowing an

exiting block to revoke changes for which it is not responsible (*e.g.*, the routine-based disabling of E). As a result, the routine-based disable does not persist beyond the exit from the block. Recall that routine-based propagation-control is needed exactly in order to affect propagation control beyond block scopes, which is impossible with restoration. This situation occurs whenever there is a routine conflict, which is exactly when conflict resolution is needed. As can be seen from their definition, change removal and restoration behave identically when there is no routine conflict. Hence, whenever they behave differently, restoration invalidates the advantages of routine-based propagation-control. Therefore, restoration is not a useful conflict resolution strategy.

2.5.9 Further Alternatives

By combining previously discussed approaches, additional solutions are possible.

Combining BSR and BER ("BSER")

As a potential fourth alternative, a mixture of BSR and BER is conceivable in which the normal _Enable/_Disable block has equal priority to routine-based propagation-control (like in BER). If the programmer wishes to enforce block awareness, a stronger block-based directive (*e.g.*, _ENABLE/_DISABLE) could be used for which the conflict resolution rules from BSR apply. There should be no conflict with the normal BER-like blocks and with regard to each other, both should have equal priority (although other designs are conceivable).

This alternative has the advantage that, most of the time, the simple semantics of BER can be used, but when there is special need to ensure that no exceptions be propagated inside a block, a mechanism exists to achieve this goal.

Of course, by introducing another form of block-based propagation-control, the overall complexity of the system increases, which may outweigh potential advantages gained from the simplicity of BER combined with the convenience of block awareness.

Two orthogonal mechanisms ("BOR")

Lastly, in order to avoid conflicts entirely, each propagation-control mechanism could apply to a different subset of exceptions. The default behaviour should be for the propagation of all exceptions to be controlled using the block-based approach since it works very well in most situations. For those situations in which routine-based propagation-control is required, a different exception type could be used whose propagation is only affected by routine-based propagation-control.

Since the two mechanisms employed here are entirely orthogonal, there are no conflicts to be resolved, and the semantics are very clear. As a result, block awareness is preserved wherever it is expected, and routine-based propagation-control can be employed wherever required.

On the downside, this approach does not allow for the same exception to be controlled by both the block-based and the routine-based approach. On the other hand, such a capability should not be required frequently, and in the rare case where it is, there is always the possibility to emulate block-based through routine-based propagation-control. Finally, introducing another exception hierarchy increases the overall complexity of the design (along with the implementation). Once there are two hierarchies with regard to propagation control, there is not much to prevent an even finer granularity of propagation-control, *e.g.*, there could be special exceptions that cannot be disabled/enabled at all, or only under special circumstances.

2.5.10 Conclusion

The preceding discussion examines the properties of finite extent in comparison to infinite extent. While the context of the discussion is asynchronous propagation control, the analysis can be generalized to other operations where different options for extent exist (*e.g.*, mutual exclusion regions). The exception is the discussion on vulnerable handlers (Section 2.5.4, p. 34), which is unique to asynchronous exceptions handling, as well as the discussion justifying that routine and block conflict are the only conflicts deserving consideration[7] (Section 2.5.5, p. 35).

Designing an EHM that supports two propagation-control mechanisms of different extent concurrently is a difficult task. There is no optimal way to resolve conflicts, only different compromises favouring different aspects.

The block-based approach offers various advantages to the user, which constitutes a strong argument against solutions like RSB. Out of BSR and BER, BSR appeals due to its preservation of block awareness and the resulting ease of programming. On the other hand, if block awareness is not essential, BER with change removal is a good alternative due to its simplicity and clear semantics, at the expense of producing some counter-intuitive results. The combination of the two previous methods in BSER is very flexible and powerful but greatly increases the complexity of the design.

[7] If, for other operations, it can be shown that no prioritization or homogeneous conflict exist, then the following analyses can be generalized as well.

The cleanest solution may be to entirely preclude conflicts between block and routine-based propagation-control. Most simply, this means just using one mechanism exclusively. However, if both are needed, conflict avoidance schemes are necessary. The only solution that precludes conflicts entirely is to have two orthogonal exception hierarchies like in BOR, with the resulting increase of complexity. If such a high complexity is undesirable, a good alternative is to use a BSR method that throws an exception or even aborts the program upon encountering a routine conflict. However, such a scheme only works if the programmer knows where routine-based propagation-control is used (*e.g.*, in a library), which requires additional documentation or syntax.

The requirement for simplicity of design may require choosing just one propagation-control extent. The choice here lies between power of expression, and safety from programmer errors. Since μC++ is employed as a teaching language, many novice programmers need to be able to use it correctly. The choice of the safer finite extent for its propagation control is therefore fitting. The resulting restriction of functionality has rarely caused problems in practice.

2.6 Implications of Infinite Scope

Interestingly, the choice of asynchronous propagation control has strong implications for the asynchrony model in which it can be used safely. In general, infinitely-scoped models can only be used safely if restricted asynchrony is in effect.

2.6.1 Infinite Scope and Full Asynchrony

To understand this restriction, suppose a language with infinitely-scoped propagation-control, *e.g.*, μC++, employed a full-asynchrony model. Consequently, due to the infinite scope of its asynchronous propagation control, any routine called within the scope of an _Enable statement must be made robust with respect to interruption at any location within its body. Consider the example in Figure 2.1, p. 23 and the call to doSomething. If doSomething is a pre-compiled routine, a programmer using it may have no way to inspect the code and verify whether it can safely be interrupted by the propagation of an asynchronous exception (*i.e.*, whether it contains vulnerable code). Furthermore, suppose doSomething is declared as not propagating exceptions or propagating exceptions other than DeadlineAlarm and ServerFailure. What are the semantics if the propagation of asynchronous exception DeadlineAlarm starts while control is inside doSomething? In addition, imagine doSomething raises an exception and during its propagation an asynchronous exception is triggered. As a result, two exceptional propagations are active

within the same execution, without an obvious way to determine which one should be propagated first or at all[8].

The re-appearance of the interruptibility problem is caused by the semi-dynamic nature of the _Enable block. While the block has static extent, it dynamically affects code that is being called from within it. In this way, there is an implicit change of environment down the call chain (in callee direction) interacting with the non-determinism of full asynchrony, which causes the interruptibility problem. In contrast, poll points, which are explicit in nature, avoid the potential interruptibility problem of the implicit _Enable since propagation can begin only at well-defined locations. Since propagation can begin only out of these routine calls, programmers can protect vulnerable code just as in the synchronous case. Pre-compiled routines that unknowingly call a poll point through an indirect method (routine pointer, virtual routine call, or dynamic replacement of a routine) can be protected by consistently using exception specifications. In this way, even if programmers do not know about the poll point, they still know (at compile time[9]) that the called function can propagate an exception, which is sufficient to ensure the call does not happen within vulnerable code. If exception specifications are not used consistently, then the (immediate or transitive) caller of such pre-compiled routines needs to disable asynchronous propagation around the call or otherwise verify that they cannot invoke poll points.

Another case in point is that pthreads, which employs asynchronous propagation control of the dynamic kind, cannot actually use its full-asynchrony model (asynchronous cancellation) safely, in general. Butenhof [But97], for example, warns that caution is required when using asynchronous cancellation: It is employable only in relatively simple code, and acquiring resources is not recommended. It is important to realize that this restriction is in large part due to the infinitely-scoped nature of pthread's asynchronous propagation control.

2.6.2 Relationship with Exception Specifications

The problematic effect of the semi-dynamic _Enable block's infinite scope is further illustrated by its relationship with exception specifications. In the example from Figure 2.1, p. 23, foo could

[8]While there are concepts of *concerted* exceptions [Iss91] and methods for *exception resolution* [CR86, RXR98b, MT02, Rin06], which try to deal with related issues and funnel multiple exceptions into one, their practicality is questionable. C++ and μC++ terminate the program when such a situation occurs, *e.g.*, an exception escapes a destructor executed during exceptional cleanup.

[9]C++ compilers, unlike Java, do not support the programmer in such static exception analysis, but it can still be performed.

be declared

```
void foo() throw ( DeadlineAlarm, ServerFailure )
```

which could be verified at compile-time (assuming C++ had static exception specification checking) to ensure there are handlers in place (up the call chain) to handle these asynchronous exceptions. This convenient interplay between exception specifications and asynchronous propagation control is no coincidence since both are mechanisms to declare possible propagations of exceptions: The former is mainly descriptive while the latter actually affects program state. Unfortunately, due to the dynamic effect of _Enable, this interplay only extends in caller-direction, not in callee-direction. A routine's exception specification describes propagations that can occur on its call and, as part of the routine's interface, is mainly for the benefit of the routine's caller. Exception specifications make no statements about how safe it is to start asynchronous propagations within a routine, which is dependent on the routine's implementation. The following examples elaborate on this issue.

Suppose doSomething is declared such that it does not propagate any exceptions, *i.e.*, throw (). From this interface declaration, there is no way of knowing whether doSomething can be safely called from within an _Enable block. Under the hopeful (and naïve) assumption that exception specifications can indeed help with asynchronous propagation control, the conservative approach is to not place such a call inside an _Enable block since doSomething is declared as not throwing exceptions. Otherwise, an asynchronous exception propagated due to the _Enable block could violate doSomething's no-throw property. Note again the difference between 'not throwing exceptions' and 'being able to propagate exceptions safely'. The throw() declaration here seems more helpful than it really is since it is quite possible that doSomething can be safely interrupted at any point, and because it handles all exceptions internally[10], it does not propagate any to its caller.

Continuing with the same example, suppose the call to doSomething is replaced by one to callDoSomething, where callDoSomething is declared throw (DeadlineAlarm, ServerFailure) and calls doSomething (see Figure 2.3). Now, while callDoSomething's interface claims to propagate only DeadlineAlarm and ServerFailure exceptions, their asynchronous propagation at arbitrary locations inside callDoSomething may leave the program in an inconsistent state. Just because callDoSomething declares it can propagate DeadlineAlarm and ServerFailure exceptions to its

[10]Realistically, such internal catching can only be performed with a catch-any handler that does not reraise, which is a very dangerous technique.

```
void doSomething() throw ();

void callDoSomething throw (DeadlineAlarm, ServerFailure) {
    ...
    doSomething();
    ...
}

void foo() {
    _Enable < DeadlineAlarm > < ServerFailure > {
        ...
        callDoSomething();
        ...
    }
}
```

Figure 2.3: Exception specifications cannot help with asynchronous propagation

caller does not guarantee that it can safely be interrupted by one in the middle of its execution.

A different issue can occur when callDoSomething is interrupted (by a DeadlineAlarm or a ServerFailure exception) while it is calling doSomething, and as a result, doSomething is interrupted. This situation contradicts the previous assumption that doSomething, according to its exception specification, cannot be interrupted safely. Note that foo cannot anticipate this issue as it may not be aware of callDoSomething's implementation or doSomething's exception specification.

It is therefore easy to see that exception specifications are a concept orthogonal to propagation control, and that fail to indicate whether asynchronous propagation is safe or unsafe. The static nature of _Enable/_Disable blocks with respect to their callers is helpful in analyzing the correctness of the program, whereas their dynamic nature with regard to the routines called within them hinders such analysis.

It is conceivable to extend a routine declaration by describing whether the routine can be safely interrupted (similar to the mechanism from Section 2.5.7, p. 41 to indicate the use of propagation control), *e.g.*,

```
void foo() _Enable < DeadlineAlarm > < ServerFailure > ;
```

which would indicate that foo can be safely interrupted by DeadlineAlarm and ServerFailure exceptions. While such syntax would indeed help calling code in understanding whether foo can be safely interrupted, it is questionable whether there are many routines that are safe to interrupt in their entirety. While pure functions can be classified as safely interruptible, most routines with side effects or that interact with their environment would more likely consist of parts that are not safely interruptible. Furthermore, if the implementor of foo is certain that it can be safely

interrupted by the exception types above, the logical consequence is to explicitly enable these exceptions over the entire routine body as in Figure 2.3 and instead declare foo as propagating the exceptions, *i.e.*,

```
void foo() throw ( DeadlineAlarm, ServerFailure )
```

Consequently, whether or not the caller of foo takes advantage of the fact that foo is safe to interrupt and adjusts its propagation control accordingly becomes irrelevant since the only possible way to take advantage of this fact would be to enable propagation over the call of foo. However, foo already enables propagation over its entire body, so the enabling through the caller is redundant. Hence, the use of additional interruptibility syntax as part of a routine declaration is of little practical use.

2.6.3 Restricted Asynchrony and Infinite Scope

As a consequence of μC++'s infinitely-scoped propagation control and this model's difficulty of safely allowing full asynchrony, μC++'s exception handling mechanism currently follows a restricted asynchrony approach, *i.e.*, it implements asynchronous exception propagation by having the propagating execution poll at certain locations (*e.g.*, before entering a monitor) for asynchronous exceptions. Polling has the advantage that control is not stolen away from the propagating execution to initiate propagation, but instead, the propagating execution voluntarily relinquishes control by polling at well-defined locations. Thus, the 'surprise factor' is eliminated, and programmers can ensure no poll point is called within vulnerable code. This implementation detail also means that while an _Enable statement conceptually enables asynchronous exception propagation within its entire (dynamic) scope, in reality, only those statements containing a (direct or indirect) call to a poll point can actually be interrupted. By choosing proper poll points, it can be ensured that propagation of asynchronous exceptions can begin only at well-defined locations, eliminating the surprise factor.

In the example from Figure 2.1, p. 23, if doSomething unknowingly triggers a poll, it may not be in a consistent state to propagate an exception when the caller of doSomething, not aware of its implementation, enables asynchronous propagation. However, if the poll points are well-documented, such a situation can only arise if doSomething calls such a poll point in an indirect way (routine pointer, virtual function call, or dynamic replacement of a routine). In these cases, since the propagation occurs through a routine call, exception specifications should be used to

document possible propagations. Polling within virtual-function calls is safe as long as the following rule is observed: Either all implementations of a given virtual routine are a poll point or none are. Still, it is clear that this reasoning is an informal method, and thus, susceptible to error. A proper choice of poll points in the EHM design is essential, *e.g.*, a wait operation on a condition variable is a good choice, whereas the assignment operator is not.

Polling can be quite efficient if used sparingly. Unfortunately, this last condition conflicts with the requirement of timely exception handling, *e.g.*, for real-time applications. If an asynchronous exception indicates an emergency, maybe a timing-related alarm, it is important that propagation of this exception, if it is safe to perform, have a low latency bound. An increase in polling frequency helps but causes additional overhead; in general, a compromise is required [Fee93].

2.7 Summary

Writing correct programs using asynchronous exceptions is difficult due to the effects of the interruptibility problem. The asynchronous control-flow introduced by asynchronous exception handling complicates program understanding and even informal verification of correctness. This chapter demonstrates that asynchronous propagation control is essential for reducing the difficulties in dealing with asynchronous exceptions. There are various models for asynchronous propagation control, characterized by different scope and extent. It is difficult to combine propagation-control approaches with both finite and infinite extent, and for the sake of simplicity, choosing only one may be required. The inability to use full asynchrony safely with infinitely-scoped propagation-control can have dramatic consequences, *e.g.*, pthread's asynchronous cancellation feature being almost impossible to use in practice. In general, full asynchrony is especially difficult to employ without suitable propagation-control, which is why its advantages in terms of intuition-of-use and performance are rarely exploited. The next chapter proposes an approach to make use of full asynchrony in a safe and intuitive manner.

Chapter 3

Combination Approach for Safe Full Asynchrony

This chapter introduces a novel approach for safely incorporating full asynchrony into an EHM using existing syntax and without abandoning restricted-asynchrony semantics. In this way, it combines the advantages of both full and restricted asynchrony, which should lead toward more intuitive and potentially better-performing code.

Restricted asynchrony/polling requires a programming style in which poll points need to be present in order for asynchronous propagation to occur. This requirement, while ensuring safety, can make it unintuitive to write correct programs that use asynchronous exceptions. While the interruptibility problem is circumvented, programmers may erroneously expect propagation to occur within a region, but, since no poll points are present, no propagation occurs. In addition, the duration between the time an exception is delivered and the time it is detected/propagated at a poll point constitutes a delay in propagation. If these restrictions are deemed unacceptable, full asynchrony needs to be employed. The advantages of full asynchrony are that poll points are not required for propagation, so the error described above cannot occur. As well, no delay introduced through polling exists between delivery and propagation, and as a side-effect, the lack of spurious polling can potentially speed up a program, at least in the non-exceptional case.

However, the disadvantage with full asynchrony is that asynchronous propagation control is the only mechanism restricting propagation. Hence, programmers wishing to write correct programs under full asynchrony, despite the restrictions listed in the previous chapter, need to use propagation control defensively, leading to a programming style in which every vulnerable

piece of code needs to have propagation disabled, *e.g.*, by enclosing it in a _Disable block. As mentioned previously, pre-compiled code whose robustness regarding asynchronous propagation cannot be verified must be classified as vulnerable, so calls to such routines need to be protected, too. As a result, source code can become cluttered with precautionary _Disable blocks whose correctness is still difficult to verify. Alternatively, since it is so difficult to master, asynchronous raise might be avoided entirely, especially by programmers whose mind-set does not embrace full asynchrony (see Section 1.5, p. 13).

In order to allow for safe full-asynchrony, I propose an alternative asynchrony model that changes between full and restricted asynchrony depending on the scope of propagation-control, and a corresponding propagation-control model with characteristics of both finite and infinite scope. Its integration with existing programs based on a traditional semi-dynamic propagation-control approach is discussed, as well as empirical results achieved when implementing this approach in a prototype.

3.1 Static Asynchronous Propagation Control

Recall that static propagation-control (see Section 2.3, p. 23) is characterized by a finite scope and finite extent, *i.e.*, the dynamic scope of the _Enable statement is eliminated, meaning its effect is limited to its immediate static block (local/static scope). In the original example from Figure 2.1, p. 23, this reduced scope implies that while statements s1 and s2 can be interrupted by the propagation of an asynchronous exception, the body of doSomething cannot as it is not statically contained inside the _Enable block. Hence, the interruptibility problem with regard to a specific _Enable block is trivially solved for all routine calls contained within it, and no routine can be surprised by an interruption for which it is unprepared.

It remains then to show that the interruptibility problem is solved for the code statically contained within the _Enable block. Logically, the purpose of _Enable is to designate the block as safe for the propagation of asynchronous exceptions. Hence, it must be clear to the programmer placing an _Enable block that its execution can be interrupted while in that static block. With full asynchrony, the programmer placing an _Enable block should be aware that code executed within it can be interrupted at any time (recall the concept of block awareness, Section 2.5.1, p. 27). It is likely that the _Enable block is placed exactly around code that is considered robust with regard to asynchronous exceptions. Since propagation is only enabled statically, routine

calls are not affected by the _Enable block any more, and as a result, analyzing a piece of code for vulnerability becomes simpler. Hence, with static-only _Enable, and assuming programmers understand their own code, the interruptibility problem is manageable even under full asynchrony without the need for excessive use of _Disable blocks.

In order for full asynchrony in asynchronous exception propagation to be safe and minimize latency, the following semantics are required. Once an asynchronous exception is detected, propagation starts the next time control reaches a static _Enable block enabling that exception. Consequently, propagation can either start immediately if detection occurs while executing inside the current _Enable block, or be deferred until control reaches a static _Enable block by regular return from a routine call or by exceptional transfer into the block.

Note that by reducing the scope of _Enable to static-only semantics, it loses its full adherence to the modularization principle, *i.e.*, taking a piece of code out of an _Enable block and replacing it with a call to a function executing the same code now has different results. However, while the modularization principle is important, there are other examples preventing modularization, e.g., references to local variables and labels, as well as the return statement. That the semi-dynamic _Enable block does support the modularization principle is a direct consequence of its (somewhat unusual) semantics, and the violation of the modularization principle by the static _Enable follows directly from its intended semantics.

3.2 Rejection of Propagation on Call Boundary

The following alternative attempt to solve the interruptibility problem using infinitely-scoped propagation-control and exception specifications is flawed but is analyzed here for completeness. The approach is based on propagating asynchronous exceptions when interruption is expected, such as when a routine call occurs. Propagation can thus start at two points in time, just before a call commences or right after a call returns, which could be implemented using call-return polling [Fee93]. From the callee's perspective, both of these are safe choices to relinquish control because either the callee has not executed yet, or it is in the process of handing over control to its caller; in both cases, it can be assumed that the callee is in a consistent state. From the caller's perspective, the interruption is not arbitrary since propagation results out of a call. This behaviour is consistent with synchronous exceptions, which propagate across routine calls. This last statement, of course, is true only if the asynchronous exception is contained in the callee's

exception specification. Hence, with this approach, if an asynchronous exception is detected that is (semi-)dynamically enabled, its propagation begins on the first return from a routine whose exception specification permits propagation of that exception. The approach can therefore be characterized as one of restricted asynchrony, where each routine that can propagate exceptions according to its exception specifications is a poll point.

This approach is either redundant or fails because it potentially interferes with the protocol between caller and callee. Suppose E is an exception whose asynchronous propagation is enabled and which is contained within the callee's exception specification (for otherwise, propagation on call boundary does not apply), *e.g.*,

```
void h() throw (E);
void g() {
    ...
    _Enable < E > {
        ...                         // trigger propagation either immediately before
        h();
        ...                         // or after call to h; same as with static _Enable
    }
}
```

If the caller and not another routine up the call chain contains the _Enable statement (as above), either the exception is detected and propagated inside the _Enable block before the call commences or (at the latest) after returning from the callee due to static _Enable block semantics alone; hence, no additional functionality is gained by propagating on the call boundary compared to the static _Enable. Suppose, now, that the _Enable block is contained further up the call chain, *e.g.*,

```
void g() {
    ... h(); ...
}
void f() {
    _Enable < E > {
        g();
    }
}
```

If E can only be used in an asynchronous context, *i.e.*, it is never raised in or propagated synchronously by the callee, then the only reason for the callee (h) to have E in its exception specification is if it is aware of an asynchronous E exception being raised at it. Consequently, the callee could contain an _Enable block to (statically) enable E, *e.g.*,

```
void h() {
    _Enable < E > {
        ...
    }
}
```

in which case static _Enable semantics cause propagation to start in that _Enable block and the additional functionality of propagation on call boundary is not needed.

If, however, the exception is already used in synchronous contexts between callee and caller, *i.e.*, h is implemented as

```
void h() {
    ...
    if ( ... )
        throw E;                    // under certain conditions raise synchronous exception
}
```

then having the asynchronous propagation of E start at the call boundary can conflict with the meaning of the synchronous E. It is likely that E in the synchronous context only occurs in certain well-defined situations, *i.e.*, its synchronous propagation implies a post-condition with regard to the protocol between h and its caller. This post-condition is likely not met when E is propagated asynchronously since asynchronous propagation implies h did not run, or ran to completion, whereas synchronous propagation would likely occur after h ran, but not to completion. Furthermore, recall that the caller, g, cannot be assumed to know about the asynchronous use of E as it does not contain the _Enable statement (in f). Thus, due to its asynchronous propagation at the call boundary, the occurrence of E may cause the caller to react inappropriately. If propagation starts before the call commences, the exception in its synchronous context may imply to the caller that some of the callee's work is completed when it is not (since the callee never executes). Conversely, if propagation starts after the call completes, the exception may imply that some of the callee's work is incomplete when it is not (since the callee ran to completion), causing the caller to redo them. In either case, an error could be introduced into the program, either through omitting an action or performing it twice, *e.g.*, signalling a condition variable.

One conclusion may be that the use of E in both synchronous and asynchronous contexts causes the problem here, and clearly, this is not a generally recommended practice. However, with a static _Enable inside the callee, the callee, well-aware of the way E is used in the synchronous context, can make sure that asynchronous and synchronous propagations do not conflict, *e.g.*,

```
void h() {
    ...
    if ( ... )
        throw E;        // raise synchronous E  => implies post-condition, e.g., signal occurred
    ...
    _Enable < E > { // h's programmer has full knowledge, and can ensure asynchronous
        ...         //   propagation of E implies same post-condition as the synchronous
    }               //   raise above; hence, caller of h cannot misunderstand asynchronous E
}
```

Hence, the same exception type can be propagated synchronously and asynchronously without disturbing the protocol between caller and callee by using static _Enable blocks as above, which is a capability the propagation-on-call-boundary approach does not support. Furthermore, this approach does not add any useful capabilities compared to the alternative static-_Enable-block approach; therefore, it is rejected.

3.3 Delayed Handling Induced by Static Propagation Control

Recall that one motivation for full asynchrony is to ensure speedy handling of asynchronous exceptions, but in order to ensure safety, full asynchrony can only be enabled within a static _Enable block. If propagation is deferred because control is not within a static _Enable block, the delay can be significant and more than when using explicit polling and infinitely-scoped _Enable blocks, *e.g.*,

```
void rec( int n ) {
    if ( n == 0 ) return;                   // terminal case
    osacquire( cout ) << n << endl;         // acquiring cout's mutex lock is a poll point
    rec( n - 1 );
}
...
_Enable {                                   // enable all asynchronous exceptions
    rec( 1000 );
}
```

Note that osacquire implicitly acquires a mutex lock to serialize printing, and the execution polls for exceptions before acquiring the lock (poll point). With semi-dynamic propagation-control, once an asynchronous exception is detected, the poll method can initiate propagation once in every recursive invocation of rec, whereas static propagation-control requires waiting until all recursive invocations return. By limiting full asynchrony to just the static _Enable block, a propagation delay is introduced while execution occurs outside that block. This delay contradicts one important motivation for full asynchrony, *i.e.*, to tighten the time bound for asynchronous exception handling, and suggests that there are situations in which full asynchrony with static propagation-control alone is not a viable solution.

3.4 Combination Approach

Recall that the infinite/dynamic-scope effects of _Enable do not cause the interruptibility problem under restricted asynchrony, *i.e.*, when exceptions are propagated at poll points (and the programmer is aware of them). It is therefore safe to combine the semi-dynamic-_Enable/restricted-

asynchrony approach with the static-only-_Enable/full-asynchrony approach to produce the following semantics for the propagation of a delivered asynchronous exception:

1. When control reaches a poll point, and if propagation of that exception is enabled dynamically, the exception is propagated immediately (restricted asynchrony).
2. If control is outside of a poll point,
   a. if the current instruction (*i.e.*, the next to execute) lies statically within an _Enable block enabling the propagation of that exception, and does not lie statically within a _Disable block nested inside that _Enable block disabling the exception, the exception is propagated immediately (full asynchrony).
   b. otherwise, no immediate action is required; continue executing and propagate when Rule 1 or Rule 2a applies.

Such an approach combines the advantages of full and restricted asynchrony, solving the delayed-propagation problem from Section 3.3.

3.4.1 Regular Return

Consider the example in Figure 3.1. For the purposes of this example, imagine three basic cases in which a propagating execution detects an asynchronous ServerFailure exception, each at a different time. The ▷ symbols represent a selected number of points in the execution at which detection of ServerFailure can occur for the purposes of this example. In case A, the exception is detected while control is statically within the _Enable block (*lines 13 and 20*). Rule 2a applies, and propagation begins immediately resulting in handling at catch clause 3 (*line 22*). In case B (*line 2*), the exception is detected inside doSomething before the lock acquisition performed by the call to osacquire(cout) (*line 4*). Since detection does not occur inside an _Enable block (or poll point), Rule 2b applies, and propagation is deferred until the call to osacquire(cout) is executed. As the resulting lock acquisition is a poll point, Rule 1 applies. The poll point is located inside an _Enable block's dynamic scope allowing asynchronous propagation of ServerFailure, so propagation of the ServerFailure exception begins, resulting in handling at catch clause 1 (*line 6*). In case C (*line 5*), the exception is detected inside doSomething after the call to osacquire(cout). Rule 2b applies again, and propagation is deferred. Immediately upon returning from the call to doSomething (*line 16*), control reaches the static _Enable block, and propagation begins resulting in handling at catch clause 2 (*line 17*).

```
1   void doSomething() {
2         ▷                                          // B (deferred until the poll)
3         try {
4               osacquire( cout ) << "This is a poll point"  << endl;
5               ▷                                    // C (deferred until return to _Enable block)
6         } catch ( ServerFailure ) {                // catch clause 1
7               // executed in case B
8         }
9   }
10  ...
11  try {
12        _Enable < ServerFailure > {
13              ▷                                    // A (immediate propagation)
14              try {
15                    doSomething();
16                    ...
17              } catch( ServerFailure ) {           // catch clause 2
18                    // executed in case C
19              }
20              ▷                                    // A (immediate propagation)
21        }
22  } catch ( ServerFailure ) {                      // catch clause 3
23        // executed in case A
24  }
```

Figure 3.1: Example demonstrating combination semantics

3.4.2 Exceptional Return

An exceptional return from a routine call deserves additional analysis. Consider the example in Figure 3.2. An asynchronous exception AsyncEx detected inside foo (*line 2*) cannot be propagated immediately (Rule 2b), and is deferred. When foo raises a SyncEx exception synchronously (*line 3*), it transfers control to catch clause 1 (*line 10*), which is statically contained within the _Enable block for AsyncEx. Since control is in a static _Enable block (*line 11*), Rule 2a applies, and the AsyncEx exception is propagated, transferring control to catch clause 2 (*line 14*). This behaviour may seem unintuitive because it interrupts the handling of SyncEx and propagates AsyncEx instead. However, note that the _Enable block is nested within the *outer* try-block. Hence, an occurrence of AsyncEx *anywhere* inside the _Enable block is intended to be handled in the outer catch clause. Therefore, if programmers find it confusing that the execution of the SyncEx handler can be interrupted by asynchronous propagations, they should structure their code in such a way that handlers are not contained within the static scope of _Enable blocks (also see the related discussion about vulnerable handlers in Section 2.5.4, p. 34).

3.4.3 Multiple Asynchronous Exceptions

It may appear that these semantics allow for multiple (asynchronous) exception propagations to be active concurrently in the same execution, *i.e.*, while one asynchronous or synchronous exception

```
1   void foo() {
2         ▷                                   // detection occurs here
3         _Throw SyncEx();
4   }
5   ...
6   try {
7         _Enable < AsyncEx > {
8               try {
9                     foo();
10              } catch ( SyncEx ) {          // catch clause 1
11                    ...
12              }
13        }
14  }  catch ( AsyncEx ) { }                  // catch clause 2
```

Figure 3.2: Example of interaction between synchronous and asynchronous exceptions

is being propagated, another asynchronous exception is detected and propagated. However, this scenario cannot occur to a greater extent as it already can with restricted-asynchrony/dynamic-enables. Recall that while detection can occur inside handler code, the execution of a handler implies the exception is not active any more (it is considered handled, see Section 1.1.2, p. 3). Hence, the only user code that is executed during propagation are cleanups (object destructors in C++[1]). In order for detection to occur in a cleanup, either a poll point or an _Enable (for fully-asynchronous detection) must be reachable within it. Hence, the danger of asynchronous propagation within a cleanup is analogous to the danger of synchronous exception propagation within a cleanup. The solution, *i.e.*, making sure that no asynchronous exceptions can go uncaught inside cleanups, is analogous to the rule (in C++) that no synchronous exceptions should go uncaught inside a cleanup[2]. While an additional concern exists (asynchronous in addition to synchronous propagation), it is due to the possibility of asynchronous propagation itself, rather than due to full asynchrony. This concern already exists with restricted asynchrony and poll points, and verifying whether a cleanup calls poll points is much harder compared to verifying whether it contains an _Enable block.

3.5 Analysis of New Semantics

This section analyzes the impact of going from a restricted-asynchrony model with semi-dynamic propagation-control to the combination approach as discussed above. For this analysis, it is im-

[1]Note that finally clauses in Java are also cleanups. However, in Java, when an exception occurs within such a clause, it replaces an existing exceptional propagation; hence, multiple concurrent propagations due to cleanups cannot occur in Java.

[2]C++ semantics terminate the program otherwise.

portant to identify two common issues (*i.e.*, common mistakes) programmers experience when using asynchronous exceptions under a restricted-asynchrony model: 1. failure to propagate an exception when one is expected, and 2. propagation of an exception when none is expected.

Failure to propagate happens when a programmer expects an exception to propagate in a certain place (*e.g.*, inside an _Enable block specifically placed for that purpose), but no poll points are encountered within that region. Experiences with μC++ users[3] suggest that programmers find it unintuitive to poll for exceptions explicitly (*e.g.*, by calling uEHM::poll), yet this idiom is often required to facilitate propagation. The addition of full asynchrony alleviates this problem since it allows for exception propagation in regions without poll points, and in exactly the location where the programmer most likely expects it: the static _Enable block. As well, by adding additional _Enable blocks down the call chain, the programmer can further decrease the delay of propagation[4].

The issue with propagation when none is expected is the source of the interruptibility problem. It occurs when programmers are unaware of possible poll points in called routines (hidden poll points). With combination semantics, it is reasonable to assume that the propagation of asynchronous exceptions inside a static _Enable block must be expected by the programmer, since full asynchrony effectively turns the entire _Enable block into a poll point. Hence, there is no hidden poll point in this case. When recompiling pre-existing code, the addition of full asynchrony can cause program behaviour to change since _Enable blocks inside that code can now cause propagation to begin at locations other than poll points. If such a case occurs and the program behaviour indeed changes as a result of adding full asynchrony, it means the program relies on specific poll-point locations for program correctness, and thus, needs to be rewritten with respect to the new semantics. A similar change would be required when giving poll-point characteristics to a construct that previously did not poll. Finally, programs that positively rely on the dynamic scope of _Enable continue to work under the new semantics as the new and old semantics are identical in this regard. In conclusion, the new semantics have a positive or no effect on new code with respect to usability issues, but may require changes of pre-existing code in certain cases.

As an alternative to the combination approach, it is also conceivable to preserve the _Enable semantics (semi-dynamic, restricted asynchrony) and use another construct, say _Enable_static,

[3]These include personal experiences of people involved in the μC++ project, as well as those of undergraduate students using μC++'s asynchronous exceptions in their assignments.

[4]Note that this technique is similar to placing explicit polls, and could very well be just as unintuitive.

to provide the additional semantics (static propagation-control, full asynchrony). This splitting approach would simplify the semantics of the _Enable block compared to the combination approach, and allow the programmer more control over what asynchrony model to enable. It would also ensure perfect compatibility with older code (ignoring the effects of a new key word). However, it is questionable whether programmers would actually benefit from this splitting approach as the overall complexity of programming would be at least as high as with the combination approach, and possibly higher. While it is possible that mistakes of type 2 (propagation where none is expected) could be reduced, it is equally conceivable that mistakes of type 1 (failure to propagate) could occur more often with the splitting approach. Overall, the combination approach represents an acceptable trade-off between the additional power of allowing full asynchrony inside the static _Enable block and the resulting programming complexity. With more power comes more responsibility: In comparison to traditional semantics, programmers need to make sure that the static _Enable block is robust with regard to fully-asynchronous propagation. This additional complexity is dwarfed by the difficulties of writing correct programs with alternative approaches such as dynamic propagation-control and full asynchrony (*e.g.*, with pthread's asynchronous cancellation). Lastly, with combined semantics, there is the possibility that programmers can intuitively use asynchronous exceptions correctly while automatically benefiting from the timely propagation of full asynchrony.

3.6 Implementation

This section describes the different options considered, efforts required, and issues encountered during the prototype implementation of the combination approach, *i.e.*, the addition of static-propagation-control/full-asynchrony, for μC++. Note that μC++ works by translating μC++ code into C++ code for the GNU Compiler Collection (GCC) [Sta] with calls into the μC++ run-time library. This set-up allows μC++ to work on a variety of platforms without the complex requirement of embedding new ideas directly within the GNU C++ compiler, but restricts how certain language features can be implemented.

3.6.1 Rules of Engagement

The initial goal for the prototype implementation was to support all platforms supported by μC++ 5.6.0[5] except for irix-mips, as that support is dropped for version 5.7.0. Later, support of the

[5]Supported platforms were linux-x86, linux-x86_64, linux-ia64, solaris-sparc, and irix-mips.

prototype implementation was reduced to just linux-x86 (see Section 3.7.2, p. 80). The latest version of GNU C++ compiler supported is 4.3.3. No modification of the GNU compiler was allowed, *i.e.*, the desired semantics had to be implemented entirely within the μC++ run-time library and through source-code transformations. The reasoning behind this restriction is that any modification of GCC creates a fork that needs to be maintained in parallel to the GCC main branch, which develops at a rapid pace. Maintaining such a fork would require more man-power than the μC++-project has and endanger μC++'s multi-platform support, which is maintained by leveraging GCC's multi-platform capabilities.

3.6.2 Simple Approach

A naïve way to implement the combination approach is to analyze the code contained within _Enable blocks and inject 'static' polls, *i.e.*, calls to the propagation routine (see Section 3.6.4, p. 72), ideally, between each instruction or at least between calls. Brosgol *et al.* propose such an approach for implementing the real-time Java specification [BHR02, §6.2].

However, the number of poll points inserted thus would be enormous in order to provide the proposed semantics and would reduce performance in the non-exceptional case significantly. Ideally, an implementation should provide the desired semantics without incurring any performance overhead when no exceptions are raised, so any such excessive-polling approaches are infeasible.

Without excessive polling, implementing these new semantics presents three major obstacles: how to identify static regions at run-time, how to change control flow upon returning into a static _Enable block, and how to facilitate asynchronous detection inside the propagating execution.

3.6.3 Identifying Static Regions

In order to find out whether an instruction is contained within an _Enable block, it is necessary to know the block's precise extent (*i.e.*, its borders). This information is trivially available to a compiler (were it aware of _Enable blocks), but since μC++ employs code transformation (rather than modifying the compiler) in order to implement _Enable blocks, it needs to be encoded explicitly through available language features. There are two basic approaches to accomplish this encoding: static and dynamic.

Static Region Identification

Encoding region extent statically has the advantage that no run-time cost is incurred when encountering a region, which is compatible with the common philosophy that exception handling should not have a run-time cost unless an exception is being propagated. Thus, a static solution is desirable. One way to achieve a static solution without modifying the compiler is to exploit the existing facilities to designate try-block regions. _Enable blocks can be transformed into try-blocks guarding the same region. For each exception E that an _Enable block enables, the corresponding try-block then has to contain a catch clause for a mirror type E'. In order to find out whether E is enabled, the underlying exception handling run-time is instructed to search for handlers of E'; a matching handler designates a matching _Enable block. This approach requires creating a copy of each exception type and maintaining it in a hierarchy mirroring the original hierarchy.

Unfortunately, extracting the required information from the run-time's personality function[6] is difficult given the available interfaces. As well, catch-all handlers (*e.g.*, catch(...)) interfere with this approach since they cannot distinguish between the real exceptions and their mirrors. Hence, this approach proved unsuitable for the implementation.

Dynamic Region Identification

It is also possible to store the required information at run-time, incurring a cost. Note, however, that μC++ already incurs a run-time cost for object creation and poll every time an _Enable block is encountered, so storing two additional addresses (start and end) fits within the current approach. The address can be calculated using GCC's computed-label (unary &&) operator and placing labels at the beginning and end of an _Enable region. Note, it is important to protect these labels from relocation by the compiler's optimizer by using assembly memory-references. Finally, μC++ maintains a stack of all propagation-control regions currently in effect, so established regions can be found efficiently (in linear time).

It may appear that only _Enable regions need to have their extents recorded as _Disable blocks do not seem to require any new semantics. However, consider the following example:

[6]The role of the personality function is, given an exception and stack frame context, to decide whether there are any actions associated with the context that need to be performed as part of its language's EHM, *e.g.*, install a handler, run cleanups, or check exception specifications dynamically.

```
_Enable {
    _Disable {
        doSomething();
        s1;
    }
    s2;
}
```

Note that the return from doSomething lies statically within an _Enable block, yet propagation is clearly prohibited by the _Disable block nested within it. When identifying whether a return from a routine call should trigger propagation, it is insufficient to query whether this return is contained within an _Enable block; nested _Disable blocks need to be considered as well. Hence, nested _Disable blocks require their extents to be recorded as well (or at least one of their borders). After returning from doSomething, exceptional propagation can occur once control leaves the _Disable block, ideally, before s2. This propagation can be accomplished by checking, upon leaving such a nested _Disable block, whether exceptions need to be propagated, and facilitating this propagation (see Section 3.6.4). Note, through this approach, leaving a nested _Disable block incurs an additional run-time cost, even when no exceptions are raised.

3.6.4 Change of Control

Since entering an _Enable block constitutes a poll point, all the necessary work is already performed in this case. The only other ways for control to move into an _Enable block are exiting a nested _Disable block (see above), catching an exception in an exception handler located within an _Enable block (see Section 3.6.4, p. 71), and returning from a call located inside an Enable block. In order to change control upon return into a static _Enable block, three steps are required. First, it is necessary to determine the current call-chain. Second, the call chain needs to be examined starting from the point of the detection up in caller direction until a return into a frame is found that is statically contained within an _Enable block enabling an already delivered exception. Finally, upon returning from that call, control needs to change such that the exception is propagated.

Call Chain

While it would be easy to use debugging information to gain knowledge about the call chain, such information is not always available, and a language feature such as the one being implemented needs to work just as well in its absence. Similarly, techniques relying on frame pointers to walk the stack are unsuitable as frame pointers are often optimized away.

The remaining option is to walk the stack using unwinding information necessary for exception handling. GCC (among other unwinding libraries) offers an interface to walk the stack through _Unwind_Backtrace, which is controlled by a *trace function*. The trace function is a call-back routine that is called after each stack frame walked, and receives the context of the current frame, as well as a user-supplied argument, as parameters.

Locating the Return

In order to locate the return point of interest, the stack needs to be walked frame-by-frame. During each step, the trace function checks whether the return address of that frame is contained within a region supplied to the trace function; if a suitable return address is found, this walking of the stack can be terminated since the return site located must be the one closest to the point of detection (see Section 3.6.5, p. 73).

The remaining question is what region should be supplied to the trace function. At any given detection point, an arbitrary number of delivered exceptions can be pending, and an arbitrary number of _Enable regions can be in effect, enclosing a return point in the current call-chain. However, for every given exception, only the closest matching _Enable region that does not immediately contain a _Disable block for the same type (see Section 3.6.3) needs to be considered since the exception should be propagated as soon as control passes through that region. Hence, for every delivered exception not yet propagated by the propagating execution, there is one such *closest-matching* region. Similarly, among all these closest-matching regions, there is one that is closer to the detection point than all others (though it may match more than one delivered exception). It suffices to supply that region to the trace function as the return point contained within it is the one encountered earliest.

Overall, the cost of determining the return location is dominated by the cost of walking the stack as this operation is the most complex. However, note that this stack walking is only necessary upon an exceptional raise, which is rare, and its cost is comparable in complexity to the cost of an ensuing terminating propagation. Furthermore, this stack-walking is only required if the closest relevant region changes between asynchronous propagations, *i.e.*, an exception is delivered whose _Enable block is even closer to the detection point than the previously closest one. It is possible that hundreds of exceptions are raised at an execution between asynchronous propagations, but the stack is only walked once, *e.g.*, when all exceptions are of the same type and the propagation-control state does not change. While pathological cases could be constructed, in

general and on aggregate, the run-time effects of such stack-walking should be negligible.

Determining the relevant exception/region is performed for each detection, and requires walking the stack of propagation-control regions for each newly-delivered exception until a suitable region is found. See Section 3.7.3, p. 83 for the performance effect of this operation.

Example

Consider the following program:

```
void h() {
    _Enable < H > {
        ...                    // F and G exceptions detected here
    }
}
void g() {
    _Enable < G > {
        h();
        ...                    // Propagation occurs here
    }
}
void f() {
    ...
    _Enable < F > {
        g();
        ...
    }
}
```

Suppose two asynchronous exceptions of type F and G, respectively, are delivered, and detection occurs while control is inside the _Enable block in h. The stack of enabled exceptions is therefore H→G→F. Starting with the delivered exception of type F, the detection routine scans the propagation-control stack from left to right, identifying the final _Enable block as the one responsible: H→G→**F**. When this process is repeated for the delivered exception of type G, the responsible second _Enable block is marked: H→**G**→**F**. Assuming no other exceptions have been delivered, the _Enable < G > block is therefore determined to be the closest to the detection site. Then, the call chain is examined, which is *detection-routine*→h→g→f (where callees point to their callers). The return from the detection routine into h does not lie within the _Enable block for G, so the stack is walked further. The return from h to g *is* located inside the _Enable < G > block. Hence, propagation needs to occur when the call to h returns into g.

Change Control in Order to Propagate

Finally, in order to facilitate the change of control that causes propagation, there are two options: code modification, and return pointer modification.

Code Modification Code modification means that once the return point of interest is located, that instruction is changed, *e.g.*, into a jump/call to a routine initiating propagation (propagation routine). At the end of the propagation routine, the replaced instruction is executed before control resumes after the return point (assuming exception propagation does not alter control flow additionally), similar to how a debugger sets break points. The disadvantage of this approach is that code replacement can be complex[7], especially if CISC architectures and multiple architectures with different instruction sets are supported (as with μC++). More importantly, code modification is not execution-specific: Any code change affects all executions running this code, as opposed to just the propagating execution, which requires disambiguation inside the propagation routine, and affects the performance of all executions. For these reasons, this approach is rejected.

Return Pointer Manipulation The alternative is to manipulate the return address directly, *i.e.*, change it to the address of code invoking the propagation routine. This technique is commonly referred to as installing a *trampoline*. For this approach to work, the return pointer needs to be stored at a mutable location, *i.e.*, the (register) stack. This means that on architectures with register windows, these register windows need to be flushed first. The advantage of this approach is that it is somewhat less complex than code modification since only one architecture-specific detail (return-address location on the stack) needs to be considered, as opposed to instruction set formats and encoding. As well, since the stack is modified directly, this approach naturally only affects the propagating execution since every execution has its own stack.

One disadvantage of this approach is that after modifying the return pointer, the resulting call-chain is invalid as the return pointer no longer points to its calling frame. This leaves the execution in a fragile state: An exception propagation that unwinds the stack could ensue, but with a broken call-chain, the exception handling mechanism cannot determine a handler. One solution to this problem is to make the frame of the propagation routine have its return pointer point to the original return location, similar to injecting an additional node into a linked list. This requires modifying the return pointer of the propagation-routine frame (or the one calling it). Note that simply restoring the modified return pointer storage (*i.e.*, remembering what location is changed and putting back the old value) may suffice on some architectures; in general, however, the location where the affected return pointer is stored could conceivably change. A consequence

[7]There are tools like *Dyninst* [BH00] that abstract away this complexity, and allow for platform-transparent insertion of code into a loaded binary.

of return pointer modification is that a previously modified return pointer always needs to be reverted back to its original value before the propagation of any terminating exception (including *synchronous* exceptions) to ensure proper handler search and stack unwinding.

A further disadvantage of modifying the return pointer is that it naturally only affects regular but not exceptional returns from a call. Recall the example from Figure 3.2, p. 63. If an AsyncEx exception is detected while executing foo, the immediately enclosing _Enable block is the closest region to the detection point, so the modified return pointer is the one for the return from foo. However, when foo raises a SyncEx exception, then control bypasses the modified return pointer[8] and instead continues inside the catch handler. A different method must therefore be employed in order to ensure that the propagation of AsyncEx starts inside the catch handler, just before any user code inside the handler is executed. Note that the same criticism does not apply to the code modification approach, in general. While it is true that modifying the code pointed to by the return pointer only affects regular returns, the same approach can be used to change the code of an exceptional landing pad (the point to which control transfers after unwinding the stack), thus, forcing a change of control on an exceptional return as well. When using the return pointer modification method, an analogous approach of dealing with this issue is to modify the landing pad encoding directly–a difficult feat. Instead, my implementation exploits the fact that a catch handler must call __cxa_begin_catch before executing any handler code. The __cxa_begin_catch routine is dynamically replaced by a routine that redoes all of the detection steps described so far, *i.e.*, re-evaluate the call chain and surrounding regions and modify the appropriate return pointer, as well as calls the original __cxa_begin_catch before it finishes.

Since any modified return pointer is reverted before a stack unwinding, no special provisions are required for cleanups run through object destructors. A destructor is a routine, and hence, while executing a destructor, control can only pass through a static _Enable block if that _Enable block is contained within the destructor itself. But establishing an _Enable block is a poll point, so propagation would need to start there in any case; care must be taken to handle a resulting termination exception within the destructor, for otherwise the program needs to be terminated as two propagations cannot be active concurrently within the same execution (see Section 2.6.1, p. 49, Section 3.4.3, p. 62).

[8] In fact, there is no modified return pointer as such a modification is reverted before stack unwinding.

Propagation Routine It remains to explain how the actual propagation is initiated through the propagation routine. Simply storing a pointer to an exception to be raised is insufficient since there could be multiple resumption exceptions that match the (current) _Enable region through which control passes. Instead, all delivered exceptions that are enabled inside that region are raised in the order they were delivered (FIFO semantics of exception propagation). If one of these exceptions has terminating semantics, this process is terminated and detection is repeated as described previously. While the propagation routine closely resembles a regular poll as employed inside a poll point, only exceptions enabled by the innermost closest-matching _Enable region are considered as opposed to all dynamically enabled exceptions.

State Restoration If detection occurs while control is statically inside an _Enable block, execution can theoretically interrupt any instruction, *e.g.*, a test/comparison. The following execution of the propagation routine can destroy the register state and leave the execution corrupted after the exception handler completes. This issue especially affects resumption semantics since it causes control to return to the exact point of interruption. Signal handlers take great care to save the register state at the point of interruption and restore it upon return. By placing an appropriate call to the propagation function inside the signal handler responsible for the interruption, its state restoration can be leveraged for the asynchronous-propagation implementation. Alternatively, the installed trampoline can save/restore the state explicitly if the architecture allows access to the entire register state as it appears at interruption. If a trampoline is installed as a replacement for a return from a function call, *e.g.*, in

```
_Enable {
    int val = foo();
}
```

the return from foo into a static _Enable block is hijacked, the trampoline code needs to save/restore return values passed through registers in order for val to contain the correct value.

3.6.5 Asynchronous Detection

It remains to be shown how asynchronous detection is facilitated. Note, detection occurring inside a poll point is *synchronous detection* and irrelevant to this discussion. Since the rules for propagation determine exactly when propagation has to occur once an exception is detected (see Section 3.4, p. 61), the only remaining variable affecting propagation delay is the frequency of detection, as well as the delay between delivery and detection (assuming that the delay between

raise and delivery is minimal, as is the case when both are combined; see Section 1.1.2, p. 3). The steps outlined in Section 3.6.4, p. 68 are part of the detection phase and best performed by the propagating execution because they depend on the propagating execution's stack contents. These stack contents could otherwise be changing while another execution tries to access them. Delivery, on the other hand, is initiated by another execution, the *delivering execution*, which is often the raising execution. In order to achieve a low delay between exception delivery and detection, ideally, detection should take place immediately after delivery. Since delivery is asynchronous, detection itself therefore requires an asynchronous transfer of control.

Context Switching

A simple approach is to attempt detection only when a propagating execution is scheduled for execution, *i.e.*, on the back side of a context switch, including those that occur as a consequence of time slicing. μC++ provides the uMachContext::restore routine as part of the mechanism to allow user code to be executed on the front- or back-side of a context-switch; it is run automatically on the back-side of every context switch. By exploiting the implementation of this mechanism, a delivering execution can indicate to the propagating execution that there are undetected exceptions. In accordance with the common philosophy of implementing exception handling, the check for newly-arrived (undetected) exceptions only incurs a run-time cost if new exceptions have actually been delivered, *i.e.*, if no new exceptions have been delivered, there is no additional overhead introduced by asynchronous detection.

Immediate Signal

It may be possible to detect exceptions more quickly if the propagating execution is running at the moment of delivery. The delivering execution can (POSIX-)signal the processor/thread on which the propagating execution is running, so the signal handler on the propagating execution's side can then initiate detection. The delay between delivery and detection would thus be dominated by whatever granularity the operating system guarantees for signal handling, and should be minimal for all practical purposes.

While the immediate-signal approach can potentially achieve the lowest practical delay, it is important to understand that the actions outlined in the context-switching approach need to be performed in any case since it can never be guaranteed that the propagating execution is running at the moment of delivery. Note, there is a race between determining the processor (kernel thread)

on which a task is executing and this task's switching to another processor. However, the only way a task can change processors is by context-switching, and detection is guaranteed to occur on the back-side of the context-switch. So while the race exists, it is benign since its only 'negative' consequence is that a task other than the propagating task can handle the signal, and thus, perform detection, which is benign.

Evaluation

One question is whether the context-switching approach alone suffices or whether an immediate-signal implementation is also needed to reduce delays. [MJMR01] mentions a periodic check for delivered exceptions in their implementation of full asynchrony, but also that a "message" may have to be sent to the target. The asynchronous nature of the communication through asynchronous exceptions argues that, since there is no synchronization between raising and propagating executions, there are few expectations the raising execution can have with regard to timely detection. Should raising and propagating execution synchronize through any of the synchronization mechanisms provided by μC++, the poll points implicit in all synchronization operations would cause detection to occur. On the other hand, it is conceivable that raising and propagating execution communicate through other means, *e.g.*, a shared variable:

```
volatile bool sent, ack, received = false;
_Task fred {
    void main() {
        for( ;; ) {
            try {
                _Enable {
                    if ( sent )     ack = true;   // acknowledge that an exception was raised
                }
            } catch ( Ex ) { ack = received = true; }  // signal that the exception is handled
        }
    }
};
void uMain::main() {
    fred f;
    _Throw Ex() _At f;
    sent = true;                // inform fred of raised exception
    while ( ! ack );            // wait until fred sees sent == true or the exception propagates
    while ( ! received ) {
        /* uMain knows that fred ran but has not yet handled the exception */
    }
}
```

In this way, the raising execution uMain knows that its exception could have been detected but was not. It is unclear whether this possibility is problematic, so in order to avoid it, immediate signalling is employed.

3.7 Evaluation of Implementation

In order to evaluate the implementation in terms of how well it provides the semantics from Section 3.4, p. 61, particularly Rule 2a, under the restrictions outlined in Section 3.6.1, p. 65, this section first examines the implementation's limitations. It then provides sample programs that demonstrate the new functionality while comparing performance/overhead to programs using the old restricted-asynchrony semantics.

3.7.1 Limitations

The implementation as completed so far suffers from the following problems and limitations.

Inlining

Optimizing compilers like GCC can inline calls to functions, *i.e.*, replace the call by the instructions contained in that function. In this way, code that is not statically contained within an _Enable block can be relocated to be nested within code that is contained within an _Enable, *e.g.*,

```
void smallFunction( DB &db ) {
    db.open();
    ...
    db.close();
}
void interruptible() {
    _Enable {
        smallFunction();
    }
}
```

Suppose smallFunction is not suitable for full asynchrony (*e.g.*, between open and close), and since it is not contained within an _Enable block, it appears not to be interruptible. However, if the call to smallFunction is inlined by the compiler, the current implementation erroneously determines that the entire routine is contained within an _Enable block, and potentially propagates between the calls to open and close.

There are various ways to tackle this problem. Ideally, the compiler should understand asynchronous propagation-control regions and make sure not to inline calls inside an _Enable block. This solution requires modification of the compiler, but when such modification is impossible (see Section 3.6.1, p. 65), workarounds have to be employed. When there is no way to influence inlining decisions of the compiler, one solution is to wrap all function bodies inside _Disable

blocks. Apart from the additional overhead for establishing the _Disable block, this solution also suffers from precluding propagations through poll points. If the compiler offers control over inlining, various approaches are possible. First, inlining could be disabled for the entire program. Obviously, this approach, while simple and effective, precludes this optimization and may therefore degrade performance. Second, the compiler could be instructed not to inline calls to smallFunction, *i.e.*, by declaring it __attribute__((noinline)) . This approach is more targeted than the previous one, but requires understanding (either by the programmer or by the μC++ translator) that inlining is a problem in this case. It also precludes inlining smallFunction at call sites that are not contained within an _Enable block. Finally, GCC 4.4 allows optimization options to be set per function definition. Thus, interruptible can be compiled with inlining disabled. This process can be automated for any routine containing _Enable blocks. Again, some optimization is precluded, namely at those call sites inside interruptible that are not located within an _Enable block. Still, the last approach currently seems to be the best compromise.

Address Migration

The implementation heavily relies on addresses obtained at run-time to determine region extent. Optimizing compilers can move code around in such a way as to invalidate the stored region addresses. While certain hints can be given to the compiler (*e.g.*, by the use of assembly memory references), there is no guarantee that it does not try to outsmart such hints. This problem applies primarily to new compiler versions, so whenever a new version appears, the prototype implementation needs to be retested thoroughly. Again, modifying the compiler to recognize propagation-control regions would ensure such issues cannot occur.

Unwind Regions

Exception handling mechanisms based on unwind tables store the extent of regions of code and the actions to be taken in these regions in statically-generated tables (*e.g.*, see [BR86, KS93, DGL95]). If a compiler has access to the complete source code, it can determine statically that a particular function call cannot result in an exceptional propagation, *i.e.*, the function does not propagate (synchronous) exceptions[9]. In such a case there is no need to create unwind regions encompassing calls of that function, and no respective table entries are stored. However, with

[9]Note, exception specifications alone may not be sufficient as their check might occur at run-time (C++), or because of unchecked exceptions (Java).

asynchronous exceptions as implemented in this work, any routine can effectively propagate an exception since propagation occurs by replacing a return (of an arbitrary function) into an _Enable region by an invocation of the propagation routine, *e.g.*, suppose in

```
void foo() {
    int i = 42;
}
try {                          // try-block appears to be guarding call to foo
    bar();
    _Enable < E >{
        foo();
    }
} catch ( ... ) {}             //  but compiler may move it to exclude call to foo
```

the execution of foo is interrupted by the detection of an asynchronous E exception. As a result, the return from foo into the _Enable block is replaced by a trampoline to the propagation routine, which propagates the E exception. However, since the compiler can determine that foo cannot raise or propagate exceptions, it could conclude that the call to foo does not propagate an exception, and thus, omit this call site from the unwind region corresponding to the try-block. When the propagation routine propagates the E exception, it appears to propagate through the call to foo, but since this call is not covered by the unwind regions, the EHM cannot find a handler for it. Hence, it is not guaranteed that an exception propagated asynchronously can be handled properly even if the function call is properly contained inside a try/catch construct suitable for the exception type. The GCC compilation option -fasynchronous-unwind-tables, which is supposed to allow for asynchronous unwinding, does not mitigate this issue in my experience[10]. A compiler's code generation therefore needs to be modified to take asynchronous exception-propagation into account. When such modification is impossible, workarounds are necessary.

The workaround involves tricking the compiler into believing that all functions can raise exceptions. This trick can be performed by injecting an unreachable throw into every function, but in such a way that the compiler cannot statically determine it is unreachable. Alternatively, by compiling with -fnon-call-exceptions, GCC considers all function containing memory references as capable of raising exceptions, which may require the injection of an artificial memory reference such as asm("" : : : "memory") into functions[11]. Both of these methods are not guaranteed to work for pre-compiled routines; however, the second option should have a higher probability of success.

[10] It is described as producing more precise unwind regions whereas less precision is needed in this case, *i.e.*, regions that encompass a greater number of call sites.

[11] Note, this assembly instruction does not have a run-time effect other than potentially precluding optimization.

A similar issue occurs when execution is interrupted while inside an _Enable block, *i.e.*, a non-call instruction is interrupted. Only routine calls can propagate exceptions[12], so it is not guaranteed that the interrupted instruction is covered by an unwind region, *e.g.*, in

```
1  try {
2      _Enable < E > {
3          int res = 1;                          // assignment
4          for ( int i = 2; i <= n; i++ ) {      // assignment, increment, conditional jump, etc.
5              res *= n;                         // arithmetic, assignment
6          }
7          cout << "Result :" << res;            // routine calls
8      }
9  } catch ( ... ) {}
```

The top part of the _Enable block (*lines 3-6*) contains non-call instructions, which cannot propagate synchronous exceptions. Only the calls to << (*line 7*) can potentially propagate an exception. The compiler could therefore conceivably optimize the generated unwind region corresponding to the try-block to encompass only the calls. Asynchronous propagation, however, can potentially occur anywhere, *e.g.*, inside the loop, and while it looks like the try-block guards it, because of the optimized unwind region, it does not, and the asynchronous propagation fails to find a handler. Testing revealed that this problem, while rare, does indeed occur. Since unhandled asynchronous exceptions terminate the program, the issue cannot be ignored or fixed easily. Therefore, the prototype implementation refrains from causing asynchronous propagation by interrupted non-call instructions, *i.e.*, asynchronous propagation only occurs when control returns from a routine call into an _Enable block. Without changing the compiler code generating the unwind regions, this limitation seems unavoidable. If the compiler can be modified, it should generate unwind regions and corresponding cleanup actions encompassing all possible propagation locations (using propagation-control analysis). Note that if propagation is possible in many locations and lots of cleanups need to be run (*e.g.*, many non-trivial automatic objects), such a solution could produce large program binaries.

Asynchronous Detection

A similar problem exists when trying to perform detection from within a signal handler. Since asynchronous detection is inherently asynchronous (as opposed to synchronous detection, *e.g.*, inside a poll point), a signal handler is part of the call chain of every asynchronous detection: Detections that are a consequence of time-slicing are ultimately called from within a timer-alarm

[12]Even the C++ throw-statement is converted into a routine call by GCC.

signal handler causing the time slice, whereas detections resulting from immediate signalling are ultimately called from within that signal's handler. Depending on where exactly the signal occurs that triggers the handler, proper unwinding information may not be available, and thus, the detection process fails since it relies on unwinding information to determine the current call-chain. In such a case, the only solution is to continue execution until a signal occurs in a more useful location. Note that the current implementation does not propagate exceptions when a signal handler is part of the call chain. Such a propagation is discouraged by the C++ standard [Int98, §18.7] and could lead to undefined behaviour.

Another issue arises from the fact that walking the stack causes the unwinding mechanism to acquire mutex locks. Such an acquisition can cause the executing thread to block, but blocking while executing a signal handler or context-switch restoration (which is when asynchronous detection happens) may not be safe in general. Although the μC++ reference manual [Buh09] does not restrict the use of restoration in this way, it is a potential issue warranting further investigation.

3.7.2 Obstacles

Development of the prototype implementation was plagued by several bugs and incompatibilities of the underlying GCC/library run-time support and/or operating systems. A major reason for these problems is that exception propagation in signal handlers is discouraged (see Section 3.7.1), and hence, this case is not tested properly or outright rejected by compiler/operating-system developers. The reason is that asynchronous unwinding is very complex since execution can be interrupted while setting up stack frames for recording the very information that enables the unwinding. The result is that walking the stack using the unwinder mechanism as the result of an asynchronous signal often does not work or does not work correctly, either because no unwinding information exists, or because it does not account for all possible interruption points. Many of these problems only manifest themselves in certain situations, or after millions of test iterations, making them especially difficult to detect, and often only after significant implementation work has been performed.

Linux-x86_64

The x86-64 architecture suffers from the following problem: Under rare circumstances, when the stack is walked from an asynchronous-signal handler using the unwinder interface, the program crashes due to a bug in the unwinder [Kri09b]. The linux-x64 support for the prototype

implementation was therefore dropped.

Linux-ia64

On the IA-64 platform under Linux, the *libunwind* library [Lib] is generally employed to provide stack unwinding functionality (other platforms often use the GCC unwinder by default). Libunwind suffers from a bug in which programs that use swapcontext and stack unwinding can crash unpredictably [Kri09a]. Since μC++ uses swapcontext for context switching on linux-ia64, and context switching is essential for μC++'s functionality, the result is that exception handling in μC++ under linux-ia64 is effectively broken. The linux-ia64 support for the prototype implementation was therefore dropped as well.

Solaris-sparc

On the solaris-sparc platform unwinding/stack-walking through a signal handler is generally not supported. However, any asynchronous detection must start below a signal handler frame (see Section 3.7.1, p. 79). Consequently, solaris-sparc support was dropped.

Linux-x86

The linux-x86 architecture is probably the most commonly deployed out of the ones supported, and the one most thoroughly tested by millions of users world-wide. It is therefore not surprising that this architecture exhibited the least severe problems in testing. However, even on this platform, unwinding/stack-walking through an asynchronous signal handler can fail on very rare occasions. Since linux-x86 was the only platform left, it could not be dropped. Instead, I devised a way to overcome this issue by skipping the problematic signal handler frames, as depicted in Figure 3.3. Here, "μC++ kernel code" means code that is executed by the μC++ kernel in order to facilitate time-slicing and/or (as in the example) asynchronous detection. The unwinder frame is the stack frame at which the stack-walker/unwinder starts. There are two signal handlers to be skipped: the one for SIGUSR2 facilitating immediate detection, and the one for SIGALRM facilitating time-slicing. Since interesting _Enable blocks are contained in user code only, it is safe to skip the kernel code. The implementation keeps track of the bottom-most user-code stack-frame, the one interrupted by the signal. The detection mechanism then uses this information to trick the unwinder interface into resuming the stack walk in that interrupted frame, as opposed to

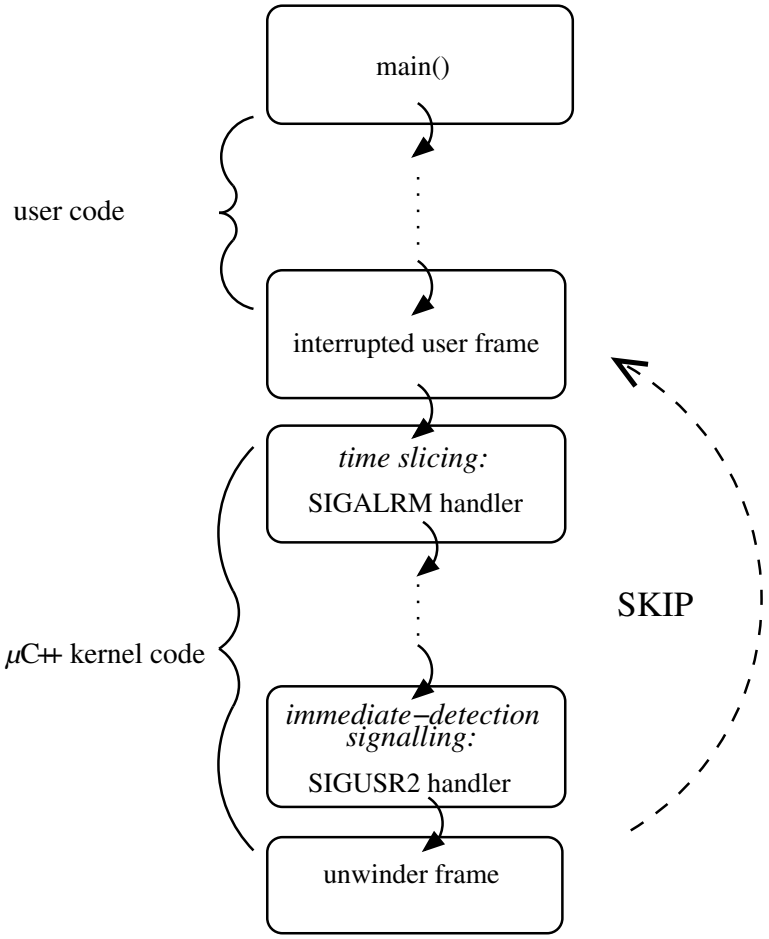

Figure 3.3: Skipping the signal handler frame during stack walking

walking through the signal handlers. Note, this kind of solution could potentially be applied on the solaris-sparc and x86_64 platforms, which was not explored due to time limitations.

On rare occasions, it can happen that the signal interrupts the program in such a state that walking the stack using the unwinder interface is impossible. In such a case, nothing can be done, and the program resumes until the next time detection is attempted (context-switch, or new delivery-signal). This restriction is unfortunate, but note that immediate propagation of an asynchronous exception is never guaranteed. First, it is possible that control spends an arbitrary amount of time outside of a static _Enable block. Second, even immediate installation of the trampoline is not guaranteed as the propagating execution may not be executing (*e.g.*, waiting on the ready-queue) when an exception is delivered. The penalty of having to wait for another context-switch is therefore relatively small.

Alternative Stack-Walking

Since many of the problems above stem from issues with the unwinder interface, I implemented an alternative stack-walker for solaris-sparc and linux-x86 that does not depend on unwinding information. However, since such solutions–while robust with regard to signal handlers–are generally fragile with respect to interruptions when stack frames are being set-up/torn-down, this solution was not pursued further.

3.7.3 Detection Performance

The program in Figure 3.4 compares the time for exception delivery and asynchronous detection between a μC++ release without and one with support for combination semantics. In order to implement full-asynchrony semantics, every asynchronous exception detection requires searching through the propagation-control stack (see 3.6.4, p. 70). The version without combination semantics (*i.e.*, only restricted asynchrony) does not perform asynchronous detection, eliminating the search, so this test primarily measures the cost impact of asynchronous detection at different propagation-control stack heights.

The only task uMain repeatedly calls a recursive routine work that establishes propagation-control regions enabling Ex at each recursion level. At the bottom of the recursion, uMain raises a Dummy exception at itself, which causes 'asynchronous' detection (uMain is synchronized with itself, of course) in the release with combination semantics. The raised Dummy exception is never enabled, and thus, there is delivery and detection, but no propagation. The time to execute the entire loop is recorded. Six different recursion levels are tested, from 0 up to 16, to account for propagation-control stacks of different lengths. Tests were performed on a linux-86 platform with 2.8GHz dual-core CPU and compiled in multi-processor mode with -O2 optimization level and no debug checks. Table 3.1 summarizes the results, where measurements are given as average (standard deviation) over 20 runs for 10,000 ROUNDS. It shows that in order to deliver and detect the same number of exceptions, the version with combination semantics takes between 7% and 73% longer. Note, the difference at LEVEL 0 is not explained by the propagation-control stack that needs to be searched as there is no propagation control at that recursion level. Rather, there is additional overhead (setting flags, marking the delivered exception) that the version with combination semantics performs for each exception delivery. Subtracting its contribution (0.53 ms) from the time measured for the combination approach isolates the contribution of the propagation-control search to the run-time cost, which is displayed in the "adjusted" row.

```
_Event Ex {};
_Event Dummy {};

void work( int ) __attribute__((noinline));
void work (int i) {
    if ( i > 0 ) _Enable < Ex > {                  // establish new _Enable block
        work( i - 1);                              // each recursion adds 1 _Enable block onto stack
    } else for ( int i = 0; i < ROUNDS; i++ ) {    // repeat ROUNDS times
            _Throw Dummy() _At uThisTask();        // at LEVEL+1 raise at this task, unravel recursion
    }
}
void uMain :: main() {
    long long start = Time();                      // start time
    work( LEVEL );                                 // adjust recursion level by LEVEL
    std::cout << (Time()-start) << std::endl;      // calculate end time
}
```

Figure 3.4: Impact of asynchronous detection on performance

	LEVEL					
	0	1	2	4	8	16
restricted	7.52 (.060)	7.50 (.061)	7.49 (.089)	7.55 (.058)	7.54 (.100)	7.50 (.069)
combination	8.05 (.070)	8.32 (.067)	8.62 (.046)	9.18 (.065)	10.37 (.067)	12.94 (.093)
difference	+7%	+11%	+15%	+22%	+37%	+73%
adjusted	0	+4%	+8%	+14%	+30%	+65%

Table 3.1: Comparison of delivery/detection performance (ms) between two μC++ releases

As expected, it roughly grows proportionally to the recursion level. A 73% performance penalty seems acceptable considering that exceptional raises are rare, the increased functionality gained with the combination approach, and that a stack of 16 propagation-control regions is likely larger than anything encountered in practice; a stack of four propagation-control regions (with a 22% performance penalty) is probably a more realistic upper bound.

3.7.4 Performance of Non-Exceptional Code and Exceptional Propagation

In order to measure the total performance effect of the combination approach, the program in Figure 3.5 records two things: 1. the cost incurred when not handling any exceptions (to gauge the performance effect of polling), and 2. the total cost of raising, delivering, detecting, propagating, and handling of exceptions. Basically, the program simulates a routine work being called repeatedly from within an inner loop, where work can be interrupted by an Ex exception. There are four preprocessor macros that control the program. As before, LEVEL defines the recursion level before an exception is raised, and ROUNDS the number of exceptions raised. ASYNC is only

```
_Event Ex {};

void work( int, bool ) __attribute__((noinline));
void work ( int i, bool raise ) {
    if ( i > 0 )
        work( i - 1, raise);
    else                                        // at LEVEL+1 and if raise==true
        if ( raise ) _Throw Ex() _At uThisTask();   //  raise exception at this task
}

void uMain :: main() {
    long long first, second, start = Time();    // start time

// first measure non-exceptional performance
#if defined( SKIP )
    int k = 0;
#endif
    for ( int i = 0; i < 100000000; i ++ ) {
// inner loop start
        work( LEVEL, false );                   // simulate real (non-exceptional) work
#if !defined( ASYNC )
# if defined( SKIP )
        if ( ++k == SKIP && !(k = 0) )          // skip SKIP polls
# endif
            uEHM::poll();                       // only poll with restricted asynchrony
#endif
// inner loop end
    }
    first = Time();
// then measure exceptional performance
    _Enable {                                   // to avoid _Enable poll-point, enable first
        for ( int i = 0; i < ROUNDS; i++ ) try {
#if defined( SKIP )
            k = 0;
#endif
            for ( ; ; ) {
// inner loop start
#if defined( SKIP )
            work( LEVEL, k == 0 );              // if SKIP, then only raise on first iteration
#else
            work( LEVEL, true );                // simulate real work, but now with raise
#endif
#if !defined( ASYNC )
# if defined( SKIP )
            if ( ++k == SKIP && !(k = 0) )      // skip SKIP polls
# endif
                uEHM::poll();                   // only poll with restricted asynchrony
#endif
// inner loop end
            } // for
        } catch ( Ex ) {}
    } // _Enable
    second = Time();
    std::cout << first-start << "\t" << second-first << std::endl;          // print times
}
```

Figure 3.5: Comparison of total exception-handling performance

defined inside the program compiled with combination-semantics support, and controls whether explicit polls are inserted into the program. If ASYNC is not defined, a poll point is inserted into

the inner loop since poll points are required in the μC++ release with just restricted asynchrony, for otherwise, exceptions are not propagated. Finally, SKIP controls at what iteration interval of the inner loop poll is invoked. The test environment is the same as for Section 3.7.3, p. 83. Measurements are given as average (standard deviation) over 20 runs with 1,000,000 ROUNDS.

As before, there is only one task, uMain, raising exceptions at itself. While it would be desirable to have a concurrent example to test truly asynchronous behaviour, the asynchrony also makes it difficult to measure meaningful data. In experiments with two unsynchronized tasks, one raising and one propagating, measurements varied widely with standard deviations of 100% and higher for the version without polling[13], so these experiments were not pursued further.

The core of the program is an inner loop consisting of a call to work, as well as additional code (in the case without ASYNC) to facilitate polling. In the first phase of the program, uMain simulates 10^8 iterations of the inner loop, and records how much time they take in total. The routine work does not raise exceptions in this phase. This first test measures the performance effect of polling as every call to poll takes time. In the second phase of the program, the same inner loop is executed again, but with work raising an exception, which is subsequently caught inside an outer loop. In total, ROUNDS exceptions are raised and handled in this fashion, and the total time for this process is recorded. Table 3.2 summarizes the results for LEVELs of 0 to 4 with SKIP undefined. For the first phase, it shows that at a LEVEL of 0, the performance of the combination approach is more than twice as fast (it takes 56% less time) as the one with restricted asynchrony. As the recursion level is increased, the impact of the poll compared to recursion diminishes, but the combination approach is still 17% faster at a LEVEL of 4.

For the second phase, it shows that the release with combination semantics takes about twice as long as the restricted-asynchrony release to raise and handle the asynchronous Ex exceptions. At first, this result may seem surprising since the combination approach is designed to minimize the delay between raise and handling of an exception. However, by polling immediately after the call to work returns, the restricted-asynchrony version propagates the exception at the soonest possible moment outside of work. No approach could possibly be faster. In practice, such an optimal situation (in terms of propagation delay) is unlikely to occur with implicit poll-points, but would require explicit polling (as in the example). The release with combination semantics needs to walk the stack at least once for every trampoline installed (see Section 3.6.4, p. 70). In

[13]These variation can be attributed to the large disparity (several orders of magnitude) between the time for propagating an exception and the lowest practical time-slice amount.

	LEVEL			
	0	1	2	4
	Non-exceptional Performance (1st phase)			
restricted	0.839 (.024)	1.298 (.022)	2.100 (.045)	3.045 (.012)
combination	0.365 (.001)	0.834 (.002)	1.433 (.003)	2.524 (.006)
difference	-56%	-36%	-32%	-17%
	Handling Performance (2nd phase)			
restricted	10.538 (.067)	10.536 (.038)	10.537 (.052)	10.558 (.054)
combination	20.290 (.063)	21.043 (.086)	21.729 (.088)	23.174 (.107)
difference	+93%	+100%	+106%	+119%

Table 3.2: Comparison of running time (s) with and without raises between two μC++ releases

order to walk the stack, it basically performs the same actions as are needed in order to propagate the exception. In effect, it therefore performs the equivalent of twice as many exceptional propagations as the version with restricted asynchrony, leading to a run-time that is about twice as long.

Under the common assumption that exceptions are rare, a performance decrease in the exceptional case seems justified by a performance increase in the non-exceptional case. From a performance perspective, the combination approach therefore delivers on its promise in this case, *i.e.*, to increase normal program performance by eliminating polling. However, while an example program such as this can give some insight into performance aspects, it is not a realistic representation of real-world code, and the exact performance effect depends on the relationship between the amount of polling and the amount of computational work performed in a real program. In particular, one could argue that in this example, the restricted-asynchrony version polls more than it needs to. In order to explore this idea, the SKIP parameter can be used to reduce the polling frequency inside the program. Table 3.3 shows the averages[14] of the metrics from the first and second phase for the combination-semantics version ("comb"), the restricted-asynchrony version without SKIP ("noskip"), and for SKIP levels from 2 to 2000, all at a LEVEL of 1. The values for comb and noskip are taken from Table 3.2, and are displayed for comparison. As polling frequency decreases (SKIP increases), the following effects are exhibited: 1. The time for the first test decreases as relatively less time is spent on polling, and 2. the time for the second test

[14]Standard deviations were omitted out of space considerations, but were at most around 2%.

	SKIP										
	comb	noskip	2	5	10	20	50	100	500	1000	2000
1st	0.834	1.298	1.153	0.943	0.900	0.874	0.869	0.874	0.865	0.864	0.863
2nd	21.04	10.54	10.69	10.79	10.85	10.91	11.17	11.62	15.05	19.31	27.84

Table 3.3: Performance (s) when adjusting polling frequency by a factor of SKIP

increases as the delay between raise and poll grows. It appears that a SKIP value between 20 and 50 gives the best compromise between high performance in the non-exceptional case and low propagation latency. Note, however, regardless of what SKIP value is chosen, the performance of the combination-semantics version in the non-exceptional case cannot be reached since there is additional overhead for polling, as well as to determine the polling interval. Furthermore, it is not a good idea to simply decrease the polling frequency by an arbitrary factor in order to improve non-exceptional performance because values of 2000 and beyond exhibit worse performance in both metrics, compared to the combination approach.

In conclusion, it is difficult to determine an optimal polling frequency for every program; programmers probably do not want to insert explicit polls into their code, nor, if they do use explicit polls, conduct a study as in Table 3.3 for every polling loop they create. Hence, the combination approach remains attractive since it can be used intuitively without considering polling frequency, and offers maximum performance in the non-exceptional case with a reasonable performance penalty in the rare exceptional case.

3.8 Summary and Future Work

By introducing the concept of tying static propagation-control with full asynchrony and combining it with semi-dynamic propagation-control under restricted synchrony, this work shows that using a more intuitive and efficient way of employing asynchronous exceptions can be achieved without sacrificing safety or correctness. The accompanying design and prototype implementation for μC++ further demonstrate the practical viability of this approach. The techniques in this chapter might be applicable to other problems involving poll-like behaviour, *e.g.*, the safe-point checks performed in garbage-collected languages such as Java and Standard ML. At the same time, the discussed limitations of the prototype implementation hint at the difficulties of trying to implement such a feature without native compiler and run-time support.

The next steps in the development of this prototype implementation require further and more extensive testing with the help of users (*e.g.*, undergraduate students) trying to accomplish actual tasks. Such testing will reveal whether the suggested usability advantages actually materialize. It is conceivable that such tests might also reveal further limitations of the prototype implementation. These and the limitations discussed in Section 3.7.1, p. 76 should then be tackled. Initially, this means solving the inlining problem by one of the methods described.

Ultimately, a production-quality implementation requires support by the compiler front-end, but most importantly, depends on correct and complete unwinding information being emitted. This information must allow a correct unwinding no matter in what state the program is interrupted, as well as unwinding tables that correctly cover an entire region of code, *i.e.*, try-blocks. Using the results from this chapter, along with more extensive usage information, it might be possible to convince compiler developers to include such functionality in their products.

Chapter 4

Asynchronous Exception Propagation in Blocked Tasks

Asynchronous exception propagation is a useful alternative form of communication among threads, especially if timely propagation is ensured. However, timely propagation is impossible for blocked threads. This chapter[1] presents an approach to transparently unblock threads to begin propagation of asynchronous termination and resumption exceptions. The approach does not require additional syntax, simplifies certain programming situations, and can improve performance.

Note, apart from the implementation (Section 4.5, p. 108) and examples (Section 4.6, p. 114) using μC++, this chapter employs a hypothetical language in its discussion. This language is C++-like in syntax and exception model, μC++-like in asynchronous-exception model, and supports a large number of synchronization and mutual-exclusion constructs. In particular, it assumes a restricted asynchrony model since this model is more common, and some of the subtle points in the discussion are best explained with explicit polling[2]. The results are still transferable to a full-asynchrony model, as the issue of how to propagate an exception inside a blocked task is independent of the asynchrony model employed.

[1] A version of this chapter has been published as [KB08].

[2] This restricted-asynchrony model is also used in the implementation section since, historically, μC++ has employed a restricted-asynchrony approach.

4.1 Motivation

Exceptional situations may be urgent (emergencies), requiring immediate propagation and handling. For the synchronous raise, this requirement is met by the immediate start of exception propagation at the raise. For the asynchronous raise, this requirement may not be met for a number of reasons, such as restricted asynchrony, propagation control, or when the propagating execution is blocked. Prior work has examined delays due to polling/restricted asynchrony [Fee93] and different forms of asynchronous propagation-control (see Chapter 2, p. 19); this chapter analyzes the issues when the propagating execution is blocked.

Clearly, propagation cannot proceed in a blocked execution, and if propagation is delayed until an execution unblocks, urgency is forfeit. This situation is especially problematic if the operation upon which the thread is blocked becomes irrelevant because of the exceptional situation, or if the exception itself implies that the operation is futile. For example, suppose the raising execution has information that a resource cannot be released in time or is deleted; hence, any tasks waiting for the resource need to be unblocked as further waiting is futile. Therefore, algorithms assuming that asynchronous exceptions are propagated immediately or at all may face a potential unbounded wait or failure.

Figure 4.1 shows three tasks synchronizing on a barrier. If task C fails, an appropriate action is to raise exceptions at tasks A and B, which can be still executing or waiting for C at the barrier (as depicted), in order to inform them of the failure so they can roll back any changes as part of backward error-recovery. However, if tasks A and B are blocked on the barrier, they cannot react to this exceptional situation. The only solution is to release A and B from their wait and allow them to handle the exception. In order to accomplish their release, task C, as part of its error handling, needs to join the barrier. However, this resolution of the problem has disadvantages. First, it mixes error handling code inside C with normal algorithmic code to fulfil the barrier protocol, which is undesirable. Second, it is complex and error-prone: While raising exceptions at other tasks is relatively easy, determining what steps of the protocol the other tasks are currently executing and figuring out how to complete the protocol is difficult. Finally, since C does not know whether A and B are already waiting at the barrier, it is possible that C itself may have to wait for A and B to arrive. Hence, C wastes time waiting instead of more profitably undoing its actions (see Section 4.6.1, p. 114 for a concrete example with time measurements).

My approach proposes that the detection of an exception for a blocked task should unblock the

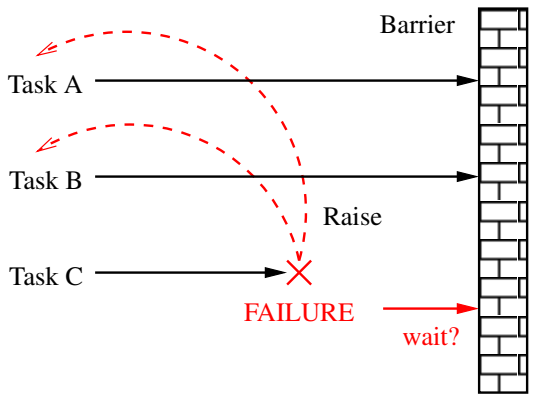

Figure 4.1: Barrier synchronization

propagating task so propagation can begin with minimal delay. This semantics follows directly from the notion of termination: Propagation aborts the current operation, so waiting for this operation should be aborted, too. The unblocking should occur implicitly and transparently, *i.e.*, neither raising nor propagating executions need to know whether the propagating execution is blocked, nor whether any unblocking occurred. With such unblocking semantics in place, task C from the example in Figure 4.1 can simply raise exceptions at tasks A and B and then continue dealing with its own exceptional situation. If tasks A and B are not blocked, the asynchronous exception forces them to react to the failure (due to task C); if tasks A and B are blocked, they are unblocked and the asynchronous exception forces them to react to the failure. While this solution appears obvious, there are significant semantic and technical issues in allowing the latter unblocking to occur in a timely fashion.

4.2 Related Work

Only a few systems attempt to address unblocking tasks in exceptional situations, but usually in a restricted or ad-hoc manner. When Modula-3 threads blocked on an AlertWait operation, which is the interruptible equivalent of a Wait, are alerted, they are unblocked, and the Alerted exception is propagated [Bir89]. Fleiner *et al.* consider the problem of exceptional unblocking, but ultimately reject it as unsafe for their pSather system [FFS96]. Ada tasks blocked on entry calls are unblocked through an asynchronous transfer of control (ATC) trigger or task abort (cancellation); tasks blocked on a protected call or entry (due to another protected action occurring on the same object) are not unblocked since cancelling the call requires executing a protected action on the respective protected object [Int95, §9.5.3 (20)]. While the Java language stan-

dard [GJSB05] does not specify its unblocking semantics, the deprecated Thread.stop (at least in Sun's implementation) can unblock threads waiting on some operations, but it does not work for all blocking operations, *e.g.*, monitor entry. Java's thread interruption through Thread.interrupt, can interrupt a thread blocked on certain (interruptible) calls, *e.g.*, wait, sleep, join, and I/O on an InterruptibleChannel, similar to Modula-3's interruption of AlertWait. This mechanism cannot directly raise an arbitrary exception, is not transparent (the interrupted thread knows it was interrupted), and threads cannot be asynchronously interrupted while executing synchronized routines. Java's java.util.concurrent package provides more comprehensive interruption support based on the lock.LockSupport.park primitive. The extension to Concurrent Haskell [MJMR01] claims to wake blocked propagating tasks in order to propagate an exception, but the reference omits the details of this mechanism. Since Haskell tasks can only block on an MVar (a kind of binary semaphore with additional information transfer), and accessing an MVar is an implicit poll point even when exception propagation is disabled, it can be assumed that the implementation exploits these poll-point semantics in order to implement unblocking. The question is how a Haskell task protects itself from unwanted unblocking. The .NET framework's asynchronous thread cancellation using the System.Threading.Abort mechanism can unblock threads blocked on a variety of operations upon *cancellation*. A cancelled POSIX thread blocked on a cancellation point is unblocked and cancellation begins immediately. Although the standard does not prescribe it, implementations exist in which a signal raised at a blocked pthread allow it to execute the signal handler despite being blocked.

Recall that thread cancellation does not constitute exception handling because the cancellation cannot be handled, and thus, only provides a very limited form of termination. Similarly, signal handling is only a crude form of resumption, and a heavy-weight feature requiring user/kernel mode switching. Furthermore, Modula-3, Java, Ada, and .NET[3] do not support resumption; Modula-3, Java and .NET do not support certain high-level concurrency-concepts, *e.g.*, Ada-style rendezvous.

4.3 Designing Unblocking Semantics for Different Instruments

Since a blocking operation usually involves a routine call, intuitively, this call should be perceived as responsible for raising the exception. Whether a potentially blocking call succeeds

[3] Visual Basic supports a form of synchronous resumption.

immediately or after some blocking is usually transparent to the programmer. Analogously, the perceived control flow should not differ between the case of an exception that propagates immediately upon calling the blocking routine, and that of the exception propagating after the thread has been blocked for some time. Neither should the raising propagation have to concern itself with whether the propagating execution is blocked. In this way, the program behaves consistently in both cases with predictable control flow, and no new syntax is required. Given the identical control flow of the blocked and non-blocked exceptional propagation, it would be useful if the local state, *i.e.*, the state of the propagating task and data it accesses in connection with the blocking operation, could be identical. The point in time immediately preceding a blocking call and in which an exception can potentially be detected (*e.g.*, through polling) shall be denoted t^-. Similarly, t^+ is the point in time immediately succeeding a blocking call when a task becomes active (after being blocked) and an exception can be detected (*e.g.*, through polling). Time t is between t^- and t^+, when a task is blocked and a pending exception is detected. Rephrasing the above design goal formally, the control flow resulting from an exception detected at t shall appear to be identical to that resulting from an exception detected at t^-. The following is an analysis and description of detailed semantics for the different scenarios in which a task can block. While the blocking instruments studied are based on those supported by μC++, the overall analysis can apply equally to any language with similar blocking instruments. Terminating semantics (throw) are analyzed first; resumption semantics are added to the design, subsequently.

4.3.1 Mutex Lock

The simplest blocking scenario is a mutex lock, which blocks the acquiring task if another task already owns the lock, *e.g.*:

```
MutexLock lock;
...
lock.acquire();                    // block if owned by another task
```

Basic Design

As discussed in Section 4.3, it is sensible to poll before (and without unblocking semantics, after) a potentially blocking call. The following example accounts for this possible exceptional control-flow before the call:

```
MutexLock lock;
try {
    poll();                          // propagation at external t⁻
    lock.acquire();                  // propagation at t
    /* critical section */
    lock.release();
} catch(...) { }                     // no need to release lock
```

Routine poll checks for pending asynchronous exceptions delivered to the execution (task), and its call[4] here represents an explicit poll at t^- before the potentially blocking call to acquire. Note, an explicit poll is shown here for illustration purposes only; in practice, a poll at t^- is performed implicitly as part of acquire. Since t^- is defined as the last point in time at which an exceptional detection is possible, the call to poll is the last possibility for exceptional detection before blocking in acquire. To simplify the example, assume that neither the critical section nor the call to release can propagate an exception. Hence, with traditional blocking semantics, it follows that an exceptional propagation can only originate in poll in the example above; if no exceptions are propagated through poll, then acquire either blocks or proceeds, but does not propagate an exception. Hence, the handler can assume the lock is not acquired.

To model unblocking semantics with the proposed semantics from Section 4.3, p. 94, the resulting control flow upon detection of an exception when blocked on acquire (at t) should appear identical to the previous case, in which propagation originates in poll at t^-. After a delivered exception is detected, a blocked propagating execution (task) is unblocked; the propagating execution can then propagate the exception, and control continues to the handler. This case differs from the previous one in that an exception can now originate inside the call to acquire. The question is whether the state of the lock is different in this case, *i.e.*, whether the lock is acquired when control reaches the handler. If the lock can be acquired, the handler may have to release it, which requires the ability to explicitly check the lock owner (not always possible). Clearly, the handler would not be the same as in the previous example because control flow in the handler depends on whether the propagating execution is unblocked, violating the transparency requirement.

To deal with this anomaly, it is necessary to define the state of the lock after exceptional propagation as follows. A call to acquire shall be defined as having failed if it returns exceptionally, *i.e.*, propagates a terminating exception. Failure implies the lock is not acquired. Hence, if a call to acquire propagates an exception, the propagating task fails in acquiring the lock[5]. This definition implies an invariant on the lock implementation: If an exception is propagated from

[4] Assume the call to poll cannot block.

[5] If the lock is an *owner lock*, *i.e.*, one that preserves lock ownership across recursive acquisitions, then the propa-

anywhere within acquire, the lock must be brought into a state consistent with acquisition failure and may require undoing any transfer of ownership (as encoded by the lock data-structure) completed so far. Conversely, if the call to acquire returns normally, it is deemed to be successful, implying lock acquisition for the calling task. In other words, acquire provides a *strong guarantee* according to [Abr98]. The proposed semantics for exceptional unblocking force a propagation from within acquire, which means acquisition failure. The result is that at t^- and t, control flows match, as well as their respective lock states. This match means transparency with regard to blocking, which indicates a sound design.

Having designed unblocking semantics for the mutex lock, other mutex/synchronization instruments of a similar nature can be treated analogously. For example, if an asynchronous exception is detected for a task waiting in the P routine of a semaphore [Dij65], the task is unblocked, P acts as the source of the exception, and the semaphore counter is adjusted to account for the unblocking task.

Poll at t^+

The discussion so far suffices to justify the design of unblocking semantics. However, for completeness, observe the effects of polling at t^+. When designing a language *without* unblocking semantics, it is a good idea to place a poll at t^+, after a task resumes execution after blocking, since exceptions delivered while the task is blocked need to be detected and potentially propagated. Whether such a poll should be placed inside the potentially-blocking acquire or explicitly by the programmer is an important consideration since the exact placement of this poll at t^+ is a subtle detail that can have complex control-flow consequences. Consider the following example:

```
try {
     poll();                        // propagation at external t⁻
     lock.acquire();                // propagation t
 →   poll();                        // propagation at external t⁺
     /* critical section */
     lock.release();
} catch(...) { }                    // no need to release lock ???
```

Here, the poll at t^- can be considered to be inside or outside acquire; propagation through this poll leads to the same unacquired lock state in both cases. However, the exact location of a poll at t^+ requires a more complex analysis. If, unlike the example above suggests, the poll is

gating task may still own the lock despite this failed acquisition if it already owns the lock from a previous successful acquisition.

contained within the call to acquire, exceptional propagation always implies failure to acquire, and therefore, no need for the handler to release the lock. Alternatively, if the poll occurs outside acquire (as indicated above), *e.g.*, it is placed there explicitly by the programmer, a propagation through this poll implies successful acquisition, and hence, the need to release the lock inside the handler. This quasi-state of the lock presents a dilemma in the handler, since the state of the lock cannot be deduced implicitly inside the handler (and possibly cannot be queried directly). Hence, the example as depicted above does not work in general and the try-block needs to be adjusted to exclude the poll at t^+.

Alternatively, the example above can be fixed by employing the *resource allocation is initialization* (RAII) technique[6] [Str94, p. 389] (see Figure 4.2). Acquiring the lock thus becomes exception-neutral, *i.e.*, if the lock acquisition succeeds, RAIIacquire's destructor automatically releases the lock after exception propagation without hindering the propagation. Hence, even when the poll at t^+ is placed outside the acquisition routine, code inside the handler can assume the lock to be unacquired. Thus, propagation in all three cases (t^-, t, t^+) leads to equivalent control flow, which is matched by equivalent lock state, regardless of where exactly the polls are placed.

The complexities of ensuring equivalent control-flow in all cases for this example argues in favour of placing the poll at t^+ inside the blocking call, *i.e.*, acquire in this case[7]. Note, when full asynchrony is supported, propagation can potentially occur anywhere it is permitted by propagation control, which implies an RAII-style acquisition is generally required to ensure exception-neutral lock release.

4.3.2 Monitor

A monitor provides mutual exclusion to a shared resource, and synchronization by blocking/unblocking tasks within the monitor [Hoa74]. This discussion also applies to task types that provide mutual exclusion and synchronization, *e.g.*, Ada's task type. There are multiple definitions for monitor semantics [How76, BFC95], and there are multiple approaches for representing the notion of a monitor in a programming language. This discussion represents a monitor as a class, first proposed in [Hoa74], with multiple public (mutex) members of which only one may

[6] While it could also be fixed by a Java-style finally clause, this would potentially require a restructuring of existing or insertion of additional try-blocks, *e.g.*, for multiple lock acquisitions. RAII can be employed without altering the block-structure of the program.

[7] Note, unblocking at t obsoletes the need for a poll at t^+.

```
struct RAIIacquire {
    MutexLock &lock;
    RAIIacquire( MutexLock &l ) : lock(l) {
        lock.acquire();              // propagation at t
    }
    ~RAIIacquire() {                 // only executes if
        lock.release();              //   constructor completes
    }
};
MutexLock lock;
try {
    poll();                          // propagation at external t−
    RAIIacquire x(lock);             // propagation at t
    poll();                          // propagation at external t+
    /* critical section */
    /* lock released implicitly at end of block */
} catch(...) { }                     // no need to release lock
```

Figure 4.2: Safe Locking/Unlocking using RAII

execute at a time. Applying the mutual-exclusion property across the public members ensures safe access to the shared data defined within the class.[8] Mutual exclusion is *implicitly* acquired and released as threads call into and return from mutex members. From within a mutex member, a task performs synchronization by calling wait on an implicit or explicit waiting queue (condition) to block, which releases the monitor mutual-exclusion so other tasks may enter, or by unblocking other waiting tasks by calling signal on an implicit or explicit waiting queue. An alternative mechanism for synchronization is to use the Ada-style accept to block until after one of a specified list of mutex members is called. Figure 4.3 shows the syntax for the two alternate styles of synchronization.

Figure 4.4 shows four possible synchronization operations in a generic monitor. The dashed lines represent blocking actions; the solid lines represent unblocking actions. The task executing in the monitor is the *owner* (black circle), and the monitor is *active* (implying mutual exclusion is acquired). Tasks calling an active monitor are blocked on the calling queue and marked with the mutex member called (*e.g.*, task b called mutex member m1). If the owner task waits in the monitor, it blocks on an implicit or explicit condition (multiple explicit condition-queues may exist) and the monitor gets a new owner or becomes inactive. If the owner signals the implicit or an explicit condition queue, there are two options denoted by signalblock and signal. The signalblock makes the signalled task the new monitor owner and the signaller task blocks on the implicit ready list [Hoa74, p. 551]). The alternative signal retains the signaller task as the

[8] Modula-3, Java, and C# also provide a lock statement for acquiring the monitor lock, which allows finer-grain locking than for an entire class member.

signal/wait

```
monitor M {
    /* shared, protected data */
    condition cv;           // queue of  waiting tasks
  public:
    void m1() {             // mutex member
        cv.wait();          // wait for signal
    }
    void m2() {             // mutex member
        cv.signal();        // signal queue
    }
}
```

accept

```
monitor M {
    /* shared, protected data */

  public:
    void m1() {             // mutex member
        accept m2;          // wait for call to m2
    }
    void m2() {             // mutex member
        ...
    } // implicitly signal task waiting for call
}
```

Figure 4.3: Alternate Synchronization Styles

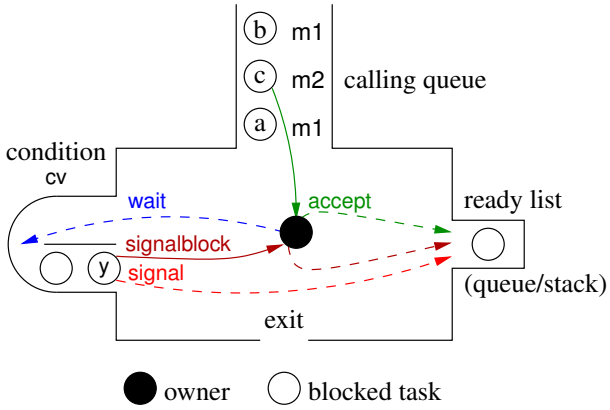

Figure 4.4: Generic Monitor Semantics

monitor owner, and the signalled task is moved from its condition to the ready list. The accept is analogous to the signalblock, but selects from the calling queue versus a condition queue using a selection criteria that searches for the first task calling a specified mutex member. For example, accept m2 unblocks task c because it is the first task waiting to call mutex-member m2, and it becomes the new owner while the acceptor blocks on the implicit ready list. If no task is calling the specified mutex-member, the monitor remains active (*i.e.*, no new owner task) until such a call is made, at which point the caller becomes the new owner. This type of synchronization is often called *rendezvous*. It is possible to specify a list of mutex members in the accept statement using multiple accept clauses, *e.g.*,

```
accept m1 {             // 1st accept clause for calls to m1
    ...                 // code executed after call to m1
} or accept m2 {        // 2nd accept clause for calls to m2
    ...                 // code executed after call to m2
}
```

which unblocks the first calling task (acceptee) that called one of the mutex members in the list.

After the acceptee waits in or exits from the monitor, the code after the accept clause is executed by the acceptor task so it knows which of the calls occurred. Before a task waits in or exits from a monitor, it attempts to keep the monitor active by implicitly unblocking a task from the ready list, and if no task is there, from the calling queue, and if no task is there, the monitor becomes inactive until a call to a mutex member occurs. This process of selecting a new monitor owner is called *monitor scheduling*.

The purpose of the ready list is to give signalled/acceptor tasks priority over calling tasks to prevent busy-waiting [Hoa74, p. 550]. Some monitor implementations violate this property by merging the ready list with the calling queue, *e.g.*, Java, C#, pthreads. The property is violated because calling tasks can *barge* into the monitor ahead of signalled tasks, which requires busy-waiting to retest the synchronization criteria.

For analyzing asynchronous unblocking, monitor execution is divided into the following categories: entry, wait/signal synchronization, accept synchronization, and scheduling after wait/exit.

Entry

A call into a monitor is similar to lock acquisition, so the same semantics can be used to achieve the desired properties, *i.e.*, unblock the task and let the propagation appear as if it originates from the call into the monitor. Because of the similarity to lock acquisition, there is consistency of control flow between t^- and t. At t^+, immediately after entering the monitor, an entering task owns the monitor, but most monitor implementations ensure that exception propagation causes it to automatically release ownership, so control flow continues just like in t^- and t, which is the same as the case for a mutex lock with RAII.

Wait/Signal Synchronization

When a task wait on a condition variable at t^-, *i.e.*, before it blocks, the task owns the monitor. Hence, if an exception is detected at t^-, it is propagated and handled by the monitor owner. For an exception detected at t, *i.e.*, after starting the wait, another task may have been scheduled in the monitor and possibly made state changes. If handling an exception by a blocked task requires monitor state-changes, the propagating task must re-acquire the monitor, which delays exception handling. It would be preferable if the exception could be propagated directly from the routine call through which the propagating task entered the monitor, bypassing part of normal stack unwinding. In this way, there would be no need to compete for monitor ownership

as propagation would start outside of it; however, such a solution is infeasible. First, bypassing violates the requirement that control at t^- and t be identical. Second, if the stack is not unwound properly between the call to wait and the entering call, handlers performing cleanups are not run, which may violate monitor invariants. Essentially, the same cleanup that occurs for a synchronous/asynchronous exception must occur for an asynchronous exception unblocking a task. Hence, the sensible design is to unblock the task, gain monitor ownership (which may require additional waiting on the monitor ready-list or some special queue), and have the exception propagate from the call to wait. The advantage of this design is that control flows at t^- and t are identical. The disadvantages are that monitor ownership and state can change between t^- and t, which can invalidate the advantage of identical control-flow. Furthermore, with the need to regain monitor ownership, there is no guarantee for timely handling of the exception. In fact, depending on monitor scheduling performed after t, the propagating task can be delayed indefinitely. Finally, by raising an exception at a task blocked on a condition variable, the raising execution implicitly influences scheduling inside a monitor of which it possibly knows nothing (also see Section 4.3.2, p. 104).

The conclusion is that a task blocked at t can at best be moved to another monitor queue by the asynchronous exception to expedite propagation, but the exception handling process cannot be accelerated further. The propagating execution must wait until the monitor becomes available before propagation can start when it is scheduled next. This restriction means the earliest the propagating execution can execute is the time when the monitor owner (at t) relinquishes its ownership.

Accept Synchronization

With accept synchronization, the analysis for a call by an acceptee is the same as that for an entry call. The analysis for the acceptor task is more complex, with separate cases when t is before or after an acceptee enters the monitor.

In the first case, *i.e.*, if the exception is detected while the acceptor is blocked waiting for the rendezvous to begin, but before an acceptee enters, *e.g.*,

```
try {                           // try-block guarding accept statement
    /* action before rendezvous */
    accept m1;                  // rendezvous with calling task
    /* action after rendezvous */
} catch( Ex ) {...}             // possibly undo acceptor's "before" action
```

the general semantics remain the same, *i.e.*, the exception detection causes the acceptor to unblock, terminating the rendezvous similarly to a rendezvous time-out [BHLC00]. Then, exception propagation begins as if originating from the accept statement and is caught by the handler, which may have to undo the *acceptor's* actions because the rendezvous did not occur. Furthermore, the acceptor task had ownership of the monitor before it accept-blocked, and since no task can have entered the monitor before the exception detection (since the rendezvous did not materialize), the propagating task can re-acquire ownership immediately. Hence, no local state change between t^- and t is possible, so consistency between propagations at t^- and t is maintained.

In the second case, *i.e.*, the acceptor is blocked during the rendezvous while an acceptee is executing a monitor call at t, the acceptor task cannot be unblocked from the ready list until it can obtain ownership of the monitor. The same analysis as in the condition-variable case applies since the acceptor task is blocked on the ready list; specifically, propagation starts as soon as the propagating task is rescheduled inside the monitor, similar to the behaviour at t^+. However, since the exception propagation now begins as if originating from the accept statement, the acceptor cannot distinguish this case from the previous one where no accept call occurred in an enclosing try-block handler. Differentiating these cases is necessary if the acceptor needs to undo the *acceptee's* actions or take some other specific action due to the asynchronous exception. Placing a try-block around the code after the rendezvous is too late because the exception propagates from the accept statement. To allow the acceptor to identify this case, a try-block specific to an accept clause[9] is given special semantics:

```
try {
    /* action before rendezvous */
    accept m1 try {        // special try-block specific for this accept clause
        /* action after rendezvous */
    } catch( Ex ) {...}    // possibly undo acceptee's actions
} catch( Ex ) {...}        // possibly undo acceptor's "before" action
```

Propagation of the asynchronous exception starts *inside* the special try-block only if the acceptee's call occurs; no propagation can take place between the accept clause and its try-block. Apart from this property, this try-block is the same as an enclosing one, and it guards against exceptions detected while blocked (at t) as well as those detected while executing code inside it (after t^+). The control flow at t is now the same as the control flow at t^+ as both see the exception arriving within an accept's try-block. It might be argued this behaviour now violates matching control flow at t^- and t. But unlike the previous cases, there is a fundamental difference here between

[9]This construct is similar to a C++ constructor try-block.

t^- and t.[10] At t^-, the rendezvous has not begun yet, whereas at t, the rendezvous is already underway and the acceptor task is blocked due to monitor scheduling. If the programmer does not take advantage of the special semantics of the specific try-block and instead guards the entire accept statement, the control flows at t^-, t, and t^+ appear identical. However, since the acceptee was allowed to execute within the monitor, a local state-change could have occurred after t^-.

Scheduling Considerations

The desire to wake tasks blocked in a monitor in order to handle exceptions is based on the goal of quick exception-handling. The minimum action is to move the propagating task from wherever it is blocked to somewhere on the monitor ready-list. The exact placement on the monitor ready-list depends on two different philosophies. One philosophy is that the exception is a phenomenon localized to the propagating task and should not affect the rest of the tasks in the monitor. Another philosophy is that the exception is the manifestation of a situation affecting the entire program, and the propagating task is merely designated to deal with it. In this view, a propagating task should probably have precedence over non-propagating tasks in monitor-scheduling decisions as it is responsible for rectifying a potential threat to the entire system. These two alternative philosophies also affect other design decisions, *e.g.*, if an exception cannot be handled, should just the propagating task be terminated or the entire program?

Regardless of the philosophy with respect to local/global effects of exception handling, any preferred scheduling of propagating tasks has to coexist with other forms of preferential task scheduling, such as task priorities. In general, the explicit prioritization of tasks is a more fundamental method of determining task precedence. Thus, a higher-priority task on the monitor ready-list should never be disadvantaged by the preferred treatment of a propagating lower-priority task; otherwise, analyses using task priorities cannot be applied any more (hard/soft real-time). However, an asynchronous exception is a communication between tasks, and it can be said that the propagating task handles the exception on behalf of the raising task. Thus, an argument can be made that the propagating task should temporarily inherit the effective priority of the raising task at the time of the raise if that priority is higher than its own. The remaining discussion assumes no or equal task priority.

If a task is both scheduled normally (*e.g.*, by a signal) and also has a pending exception propagation, then there are two possible orderings at which it can be scheduled: the one prescribed by

[10] This difference also exists for waiting/signalled tasks; however, there is no resulting difference in control flow.

the normal schedule, and the one prescribed by exceptional unblocking (the *exceptional scheduling*). A propagating task should never be disadvantaged in its scheduling order (with regard to its normal scheduling order) because of an exception; hence, the correct design is to choose the earlier between these two orderings.

The following paragraphs examine possible exceptional scheduling orders. There are three basic strategies for incorporating propagating tasks into the monitor scheduling: promoting, demoting, and neutral. The promoting strategy gives propagating tasks precedence over normal tasks, *e.g.*, by placing them in a special list scheduled before any other. Conversely, the demoting strategy schedules any other task before a propagating one, *e.g.*, by placing a propagating task at the end of the monitor ready-list. The neutral strategy treats propagating and normal tasks alike, in principle. Still, there are various nuances to implement such a strategy, and thus, bias the scheduling on a subtler level. The simplest neutral strategy is to treat the propagating task as if it had been signalled at the moment of detection, *i.e.*, it is moved to the front of the monitor ready-list. This approach is equivalent to signalling the propagating task, which subsequently polls for asynchronous exceptions after waking up. However, this *exceptional signal* is sent by the raising task, which may not be the monitor owner. Note, while some implementations allow signalling without ownership from outside a monitor, *e.g.*, POSIX threads, many monitor implementations do not.

In the demoting strategy, the time until a propagating task can execute is *at least* that of a normal signal with a subsequent poll by the waking propagating task, but likely more due to other tasks' being scheduled ahead. With a neutral strategy, other tasks can be scheduled ahead of the propagating task, which may result in no speedup of exceptional propagation compared to a normal signal and poll. With a promoting strategy, propagating tasks are scheduled at the earliest possible time, *i.e.*, when the current monitor owner relinquishes ownership. However, this scheduling makes it more difficult to reason about the order of execution after successful synchronization since asynchronously arriving exceptions perturb the normal scheduling order. Nevertheless, this effect cannot be avoided if the goal of the promoting strategy is to favour propagating tasks over the normal ordering for timely execution. Note, scheduling perturbation is only noticeable in monitors with well-defined scheduling order; monitors with no strict ordering, *e.g.*, when barging is allowed, have nothing to perturb.

Choosing the right strategy depends on which compromise is preferred between predictable scheduling order and quick exception-handling. Using a demoting strategy, the normal schedul-

ing order is preserved, but the handling of the exception can be delayed indefinitely. Conversely, a promoting strategy sacrifices predictability in favour of quick exception-handling. Neutral strategies are the least useful because neither scheduling order nor speedy exception-handling are ensured. If the philosophy is to sacrifice scheduling order to expedite exception handling, the following assumption is helpful. The propagating task, aware of its potential interference with normal scheduling order, should not manipulate the monitor beyond necessary cleanups (including maintaining monitor invariants) and leave quickly. Hence, the actual time in which a propagating task interferes with synchronization and the extent of this interference, *i.e.*, manipulation of shared data, can be minimal. For these reasons, a promoting scheduling strategy is employed in the subsequent implementation (see Section 4.5, p. 108).

It is possible for multiple asynchronous exceptions to be raised at the same or different propagating tasks blocked inside a monitor. If there is only one propagating task, then the first exception detected causes it to be promoted/demoted, and the other exceptions are processed in regular fashion after handling the first exception. Any additional exceptions cannot promote/demote the propagating task further. If there are multiple propagating tasks blocked inside the same monitor, what scheduling should be employed? In general, the relationship among these concurrent exceptions/propagating tasks is unclear. While exception hierarchies provide some notion of ordering, concurrent exceptions are not always related through a hierarchy, nor does it define an ordering for instances of the same exception class, and in any case, it is questionable whether hierarchical relationships imply any sensible scheduling preferences. Another possible ordering is the delivery order of the asynchronous exceptions. Some may claim the first exception raised is the most important one (and thus, its target the first to be scheduled) as subsequent exceptions may be symptoms of the first exceptional situation. Even if this preference is accepted, the asynchronous nature of the raise means the first exception delivered need not be the first one raised. Thus, temporal ordering by delivery is of questionable use; as well, temporal-ordering by raise requires time-stamping, and it is unclear whether this additional effort is justified or possible. My attitude is to leave the scheduling order in such a situation undefined, as is often done in a concurrent environment: A propagating task is guaranteed to be unblocked, but it is unknown in what order it executes compared to other unblocked propagating tasks.

4.4 Resumption Semantics

Resumption semantics allow control flow to return from the handler to the detection point. The resumption model is basically a dynamic routine-call (*i.e.*, the routine name is looked up dynamically versus statically). Resumption is the lesser known model of exception handling (termination being the well-known form) and its relevance has been debated [LS79, p. 549];[Str94, p. 392]. Nevertheless, resumption can be a useful form of exception handling [Goo75, Geh92, BM00, Don01]. This section completes the design for asynchronous unblocking in an environment including resumption.

For resumption, the main goal is to ensure consistency of control flow between t^- and t. Furthermore, resumption must be consistent with the semantics discussed in the previous sections dealing with terminating exceptions, especially since a resumption handler can choose to (re-)throw an exception. If an exception is detected at t^-, the resumption handler is executed before a blocking call:

```
try {
    // resumption exception at t^-
    // blocking call
} _CatchResume( Ex e ) { ... if (...) throw; ... }      // resumption handler, rethrow e
} catch( Ex e ) { ... }                                  // termination handler
```

Ultimately, the resumption handler must exit either by a (re-)throw or return. If the handler (re-)throws, propagation unwinds all handler frames until it reaches the point of the resumption detection (return point), from which point control flow is indistinguishable from a termination exception detected at t^-. If the handler returns, the task proceeds after the detection point and issues the potentially blocking call. Similar behaviour is required at t. To fulfill this requirement, the task must unblock and execute the resumption handler. However, unlike with termination semantics, the task is still conceptually attached to the mutex/synchronization instrument and must therefore remain *pseudo-blocked* on it. Hence, even though the task is scheduled for execution, it has neither acquired mutual exclusion nor synchronized, so any lock accounting information, such as a semaphore counter, is maintained as if the task is still blocked.

Formally, if the handler (re-)throws, the behaviour past the detection point shall be identical to terminating semantics. If the handler returns, resumption semantics require the task to proceed from the point of detection. At t^- proceeding from the point of detection means blocking on the mutex/synchronization instrument; analogously, at t, the propagating task must be *returned* to the blocked state at the location where it blocked originally. The safety of this *reblocking* relies on

the fact that the original blocking call did not proceed beyond t. However, this semantics requires the reblocking task to retest the synchronization criteria of the blocking instrument, possibly foregoing the reblocking, *e.g.*, if the synchronization/mutual-exclusion protocol completed while executing the resumption handler.

The following example of waiting on a condition variable shows why reblocking after executing an asynchronous resumption without re-evaluation of the synchronization criteria is insufficient. It also demonstrates why a resumption should not unblock a task fully but instead keep it pseudo-blocked. Since signals are not remembered in a condition variable, *i.e.*, there is no counter as for a semaphore, a common idiom is for the signalling task to set a flag in the monitor so a waiting task can determine whether it should wait:

```
monitor Mo {
    bool flag;
    condition cv;
  public:
    Mo() : flag(false) {}
    int maybeWait() {                                   // task A
        try { ... if ( ! flag ) cv.wait(); ...
        } _CatchResume( Ex ) { ... }                    // return to wait
    }
    void wakeup() { flag = true; cv.signal(); }         // task B
};
```

Assume task A calls maybeWait and waits on condition cv because the flag is not set. Suppose task B now calls wakeup, acquires ownership, and immediately thereafter, a delivered resumption (raised from outside the monitor) for task A is detected. As a result, A is dequeued from cv and put on the ready list due to promoting scheduling (see Section 4.3.2, p. 104). Task B continues inside wakeup, sets the flag, signals cv, which is empty so the signal is lost, and leaves. Propagating task A is now scheduled and executes its handler, and, upon return, rewaits on cv. If this rewait just blocks, task A does not realize its signal has occurred but been lost. While there are explicit programming approaches to solve this problem, it is easiest to have the condition variable implicitly manage the propagating task while it is pseudo-blocked. Hence, when the propagating task tries to rewait, the signal is associated with it, and the propagating task is dequeued instead.

4.5 Implementation

Several challenges need to be addressed in the implementation of asynchronous exception handling by blocked tasks. μC++ is well suited for this demonstration because it supports both termination and resumption, as well as bound-object matching [BK06], along with a variety of

blocking instruments of varying complexity. Basically, all the design issues presented for asynchronous unblocking are present in μC++, so it provides a challenging environment to express and solve these issues. The derived solutions are equally applicable to other languages with similar exception-handling mechanisms and concurrency facilities.

4.5.1 General Steps

The first challenge is providing for propagation, which is achieved by inserting implicit poll-points into the code before blocking and after unblocking. Inserting a poll point before a blocking operation (at t^-) makes sense since it is illogical for a task to block with exceptions in its queue that would cause it to become unblocked (at t). Such a poll must be performed in an atomic fashion, *i.e.*, once the poll determines that no outstanding exceptions can be propagated, the task must block before any new exception is allowed to be delivered to it. To facilitate detection at t, another poll is inserted into the blocking routine and placed such that it is performed immediately after a task unblocks to force propagation of the exception before further advance.

The second challenge is to determine whether the propagating task is blocked since additional action (beyond delivery) is only required in the blocked case. This check and any ensuing actions, *e.g.*, determining the suitability of the exception for propagation (by checking asynchronous propagation control), and unblocking the propagating task, must be performed by some active task. The *delivering task* (in μC++, the raising task) is an obvious choice for performing these actions because it is active and already has to manipulate the propagating task as part of exception delivery. Note, this is a case where *another* task detects exceptions on behalf of the propagating task. The information about a propagating task's running state is maintained in the task itself and can be checked easily, but the difficulty is avoiding the race condition inherent in this check, and performing a subsequent action based on the result of the check. The solution is to have a delivery lock broadly guard the blocked-check and all subsequent operations by the delivering task, as well as the poll and blocking by the propagating task. This method ensures that once an asynchronous exception is being delivered, the propagating task cannot block (except on the delivery lock). It also ensures that a task cannot block with an exception suitable for propagation on its delivered-queue. When the owner of a blocking instrument completes its synchronization/mutual-exclusion protocol with the propagating task, *e.g.*, by releasing a lock, or signalling the condition the propagating task is blocked on, the propagating task shall be designated *released*. Such a releasing operation also constitutes a race with the delivering task trying

to unblock the propagating task. Therefore, the owner of the blocking instrument needs to acquire the delivery lock before releasing the propagating task. All three tasks (raising task, propagating task, mutex owner/releaser) must agree over whether the propagating task is running or blocked. Thus, its transition from t^- to a blocked state (potential t) and the transition from blocked to t^+ constitute linearization points [HS08, §3.5.1].

The third challenge is how to activate a blocked task in order for it to process its exception queue. If the task is spinning on a spin lock, it can check its own exception queue or some flag that is set by the delivering task, and no further action is required by the delivering task. For blocking instruments, the delivering task needs to actively perform some administrative action in order to unblock the propagating task, *e.g.*, moving the propagating task off some waiting queue and onto a ready list. The propagating task should provide a method or at least additional information that the delivering task can use to unblock it (since the propagating task knows on what instrument it is blocked at that moment). Most likely, the delivering task has to acquire some lock protecting the internal data structures of the blocking instrument, which may force it to block, but these locks are usually designed to be acquired for a short time only. The resulting scenario with three tasks (raising task, propagating task, mutex owner/releaser) manipulating two locks (delivery lock, instrument-specific protecting lock) makes for a complex protocol as not all tasks can acquire these locks in the same order. In general, the waking and unblocking of a task due to an exception is similar to a time-out; hence, if time-out facilities exist, they might be exploited for the exception case. The following is a description of the detailed μC++ implementations with regard to the respective blocking instruments.

4.5.2 Mutex Lock / Monitor Entry

For any blocking lock, it suffices to acquire its (internal) protecting lock and make the blocked task ready. As the propagating task polls implicitly after waking (at t), it must detect the delivered exception and begin propagation, processing one termination or all resumption exceptions, before entering the critical region. Simple monitor entry is implemented similarly.

4.5.3 Monitor Condition-Variable

For tasks waiting on a monitor condition-variable, the implementation is more complex since the propagating task needs to compete for monitor ownership. As pointed out in Section 4.3.2, p. 98, monitor semantics vary, resulting in different implementations, and these implementation details

determine how the propagating task is awakened. For a μC++ monitor, only the monitor owner can access the various internal conditions/ready-list, *i.e.*, these data structures do not have separate locks but are collectively protected by one lock called *monitor lock*. To facilitate the scheduling of propagating tasks, the current monitor owner needs to be made aware of their presence. If no monitor owner exists, the delivering task itself must enter the monitor and perform the necessary actions. To avoid a race condition for detecting monitor ownership, the leaving/waiting owner has to perform a protocol with a potential delivering task, which requires additional locking of the monitor lock. The required action, *e.g.*, transferring a propagating task to the head of the monitor ready-list, needs to be encoded in an *action queue* the monitor owner processes when relinquishing ownership. The delivering task executes the protocol in Figure 4.5. Subsequently, the propagating task needs to poll for exceptions as soon as it wakes up.

Since the delivering task cannot manipulate the internal monitor queues directly (unless it owns the monitor), and it would be inefficient for the delivering task to wait until it can own the monitor, it needs to communicate the desired scheduling to the monitor owner. The action queue is an efficient mechanism for this communication since it needs to be processed only once and only upon relinquishing ownership of the monitor, which is the time at which the monitor owner makes scheduling decisions in any case. The disadvantage is that it complicates the implementation of certain neutral scheduling strategies. For example, consider the neutral strategy in which an asynchronous exception-detection is interpreted as an exceptional signal, moving the propagating task to the start of the ready list. However, the raising task is not the owner, so this action is deferred and recorded in the action queue. Then, the current monitor owner signals condition variables, moving tasks onto the ready list. So when ownership is relinquished and the action queue is processed, precise temporal information about when the exception was detected with respect to the signalled tasks is unavailable (or needs to be recorded/recovered); hence, replicating the scheduling order required by this neutral strategy is difficult. However, neutral strategies produce the least useful scheduling (see Section 4.3.2, p. 104). As well, this strategy tries to enforce an ordering that is, due to the asynchronous nature of the exception, inherently non-deterministic. Hence, no advantage can be gained by following this strategy, and thus, precluding its use due to the action-queue implementation seems acceptable.

- Acquire delivery lock, queue exception on propagating task P, check if P is blocked,
- if P is not blocked, release delivery lock, done.
  - otherwise verify exception's eligibility for propagation,
  - if exception is disabled, release delivery lock, done.
    - otherwise acquire monitor lock, check if P is released,
    - if P is released, release all locks, done.
      - otherwise check for a monitor owner,
      - if there is an owner, add action to action queue, release locks, done.
        - otherwise execute actions, release locks, done.

Figure 4.5: Algorithm for delivering task when unblocking a task blocked inside a monitor

4.5.4 Accepting

Rendezvous using accept (_Accept in μC++) is implemented similarly to the monitor condition-variable, with the following additional considerations. If the rendezvous has not occurred, the acceptor can simply be unblocked, and it immediately regains monitor ownership. If a rendezvous has occurred, the acceptor must be on the monitor ready-list, and either no further action is required, or the acceptor must be moved to a preferred queue to provide a promoting-scheduling strategy.

4.5.5 Resumption

Supporting resumption adds more implementation complexity. Unlike termination, which occurs once and always aborts the blocking operation, multiple different resumptions can occur while a task is blocked, resulting in multiple transitions between running pseudo-blocked and reblocking (one for each resumption exception handled). Hence, polling and subsequent reblocking may repeat when a propagating task is awakened via a resumption. Pseudo-blocking can cause further complications, *e.g.*, when an acceptee arrives while the acceptor is running pseudo-blocked. Resumption can also allow a task to re-enter a monitor, and this task must only acquire monitor ownership once along the entire pseudo-blocked re-entry chain. Furthermore, a task can block again on a condition variable on which it is still pseudo-blocked, or accept a member while pseudo-blocked on an _Accept statement (or any arbitrarily complex combination/repetition of these situations). While such program logic does not seem advisable, it cannot be rejected, and

thus, needs to be addressed by the implementation. Additional information indicating whether tasks are blocked normally or pseudo-blocked is therefore required. Then, a propagating task's transitioning from blocked to pseudo-blocked merely requires it to be designated schedulable, and, if required, added to the monitor ready-list according to the scheduling strategy. If a pseudo-blocked task is released (*e.g.*, its condition variable is signalled, or it obtains a lock or monitor), this changed status needs to be recorded as well, implying a three-state flag encoding normal, pseudo-blocked, and released status. Then, the propagating task is dequeued from its blocking instrument. After returning from the handler, the propagating task checks the flag, sees that it has been released in the meantime, and so does not reblock but simply proceeds. Otherwise, if the task has not been released by the time its handler completes, it remains queued; it reblocks and the pseudo-blocked flag is reverted to normal. Distinguishing between a new blocking operation (*e.g.*, entry, wait, _Accept) and a reblocking can be achieved by associating a stack-allocated object containing the pseudo-blocked flag for each unique blocking operation by a task.

4.5.6 Cost

Accounting for unblocking semantics incurs a run-time cost when potentially blocking operations are invoked. Table 4.1 presents a few standard performance metrics for a μC++ release without unblocking functionality, and Table 4.2 presents the μC++ release (forked off from the former) implementing unblocking functionality. Measurements were performed with μC++-5.6.0 in multi-processor mode with -O2 optimization level, no debug checks, and run on a 4x dual-core 2.6 MHz linux-x86_64 machine. The numbers represent the average (over 10,000 iterations) run-time cost of a single operation in nano-seconds. Each measurement was run 10 times, and the minimum value chosen. Since measurement noise tends to slow things down rather than speeding them up, this choice provides a good representation of the true value. The range between minimum and maximum measurement clustered around 15%, so only performance differences (between unblocking and no unblocking) greater than 30% are considered here. The operations measured are automatic object-creation (auto), dynamic object-creation (dynamic), routine/entry call with parameter passing (call), accept cycle with parameter passing (accept), suspend/resume cycle (suspend), and wait/signal cycle (wait), as well as a context-switch (cxt sw). A cormonitor is a coroutine with monitor properties.

Noticeable differences are in the entry call, where the unblocking release incurs a 30-40% penalty, the accept, where it incurs a 40-50% penalty, and the signal/wait cycle, where it incurs

Operation / Instrument	auto	dynamic	call	accept	suspend	wait	cxt sw
coroutine	162	199	14	N/A	90	N/A	
monitor	120	164	**83**	**388**	N/A	**161**	
cormonitor	249	344	**82**	**388**	87	**159**	
task	580	626	N/A	**385**	N/A	**159**	
							138

Table 4.1: Cost (ns) without unblocking functionality

Operation / Instrument	auto	dynamic	call	accept	suspend	wait	cxt sw
coroutine	149	182	14	N/A	89	N/A	
monitor	123	174	**108**	**574**	N/A	**248**	
cormonitor	228	327	**113**	**577**	88	**240**	
task	654	733	N/A	**541**	N/A	**224**	
							120

Table 4.2: Cost (ns) with unblocking functionality

a 40-55% penalty. It is unsurprising that these operations are affected since they need to be augmented extensively in order to implement unblocking functionality.

Considering the additional functionality and ease of use gained through unblocking functionality, as well as the considerable complexities in its implementation, these performance penalties seem fair. Furthermore, depending on the program, they may be more than outweighed by the faster performance these new semantics allow in certain cases (see Section 4.6).

4.6 Applications of New Feature

Two aspects of the new language feature are important: what are the effects in terms of power of expression and ease of use, and what are the effects on run-time performance? These aspects are evaluated in this section by providing sample programs and scenarios in which the new semantics are useful.

4.6.1 Worry-Free Synchronization

Figure 4.6 shows a server task asynchronously providing a computationally expensive service to a number of clients. With unblocking semantics, the lines marked with the "NoUnblocking semantics" comment can be removed. When there are no unblocking semantics, these lines must remain for correctness. Client and server follow a simple protocol: a client starts

```
#define ms * 1000000
_Event CompError {};

_Task Server {
    Client *c;
    int run, result, req;
    int compute( int req ) throw( CompError ) {
        _Timeout( uDuration( 0, 50 ms ) );          // pretend to work
        if ( ++run % 3 == 0 ) _Throw CompError();
        _Timeout( uDuration( 0, 50 ms ) );          // pretend to work
    }
  public:
    Server() : run( 0 ) {}
    void sendRequest( int n ) {
        c = (Client *) &uThisTask();                // address of calling task
        req = n;
    }
    int getResult() {
        uEHM::poll();                               // NoUnblocking semantics
        return result;
    }
    void main() {                                   // thread starts here
        for ( ;; )
            try {
                _Accept( ~Server ) { break; }       // terminate loop when destructor called
                or _Accept( sendRequest ) {         // rendezvous with client
                    result = compute( req );        // service client request (overlap with client)
                    _Accept( getResult );           // wait for client to retrieve result
                }
            } catch( CompError ) {                  // requested computation failed
                _Throw _At *c;                      // asynchronous raise at client
                _Accept( getResult );               // NoUnblocking semantics
            }
    }
} server;                                           // create server task and start running

_Task Client {
  public:
    void main() {                                   // thread starts here
        for ( int i = 0 ; i < 20 ; i += 1 ) {       // make N requests from server
            server.sendRequest( i );                // rendezvous with server
            try {
                _Enable {                           // enable propagation after try-block in place
                    _Timeout( uDuration(0, 150 ms) );  // pretend to work (overlap with server)
                    int res = server.getResult();  // attempt to obtain result from server
                }
            } catch( CompError ) {}                 // server computation failed, ignore
        } // for
    }
};

void uMain::main() {                                // program starts here
    uProcessor p[2];                                // create kernel threads
    Client c[4];                                    // create client tasks and start running
}                                                   // implicitly join with clients when finished
```

Figure 4.6: Client/Server

a computation by calling Server::sendRequest, which retains the client's id and request for asynchronous processing while the client does other work; a client calls Server::getResult to obtain the result. Note, while the processing is provided asynchronously, calls to Server::sendRequest are synchronous, and hence, may block the client until the server can begin processing its request. Assume some client inputs are faulty, which the server detects in half the time it takes it to perform a computation, aborting the computation. The server's catch clause handles the faulty case (CompError) by relaying the exception asynchronously to the responsible client, which may be working or waiting for the result. Without the new unblocking semantics, the client cannot respond to the exception if it is blocked waiting for the result. Hence, it is necessary for the server to complete the synchronization protocol by accepting Server::getResult in the handler (marked with the "NoUnblocking semantics" comment), even though there is no result, so the client can unblock and propagate the exception. With the new semantics, this additional call is unnecessary as the exception wakes the client. As well, without the new semantics, it is necessary for the client to poll at the start of Server::getResult and subsequently receive the CompError exception in order to ensure it does not return an arbitrary result and proceed to use it. This explicit poll is unnecessary with the new semantics. To summarize, as soon as asynchronous exceptions influence control flow, synchronization becomes more complicated without the new semantics as special precautions need to be taken when a propagating task may be blocked. With the new semantics, the programmer need not worry about the intricacies of synchronization under exceptions, and no extra code is required as the raise automatically does the right thing. As a side effect, the program in Figure 4.6 also becomes more efficient because blocking due to additional synchronization is avoided: without the new unblocking semantics, the runtime is between 12.01 s and 12.02 s; with the new semantics, the runtime is between 9.40 s and 9.41 s (10 runs each).

4.6.2 Cheat and Run While Blocked

Figure 4.7 consists of a number of Worker tasks, each operating on a distinct portion of data. To calculate a result, a worker does prework independently and then completes the work inside a common monitor. However, some of the values supplied to the workers are erroneous. The main task therefore sends out messages (as resumptions) to revoke the faulty values and trigger a (re-)calculation. Note, this situation is a natural application for asynchronous resumption as the correcting action is an independent interruption with subsequent continuation of the execution path of a worker task.

```
#define ms * 1000000
const int TASKS = 8, CHUNK = 10;

_Event Recall {
  public:
    int i, correction;
    Recall( int i, int c ) : i(i), correction(c) {}
};

_Monitor M {
  public:
    void work( int &i ) {
        _Timeout( uDuration( 0, 100 ms ) );        // final work done serially
    }
} complete;

_Task Worker {
    int *chunks;                                   // chunks for computation
    void prework( int &chunk ) {
        _Timeout( uDuration( 0, 250 ms ) );        // prework done independently
    }
  public:
    Worker( int chunks[] ) : chunks( chunks ) {}
    void main() {                                  // thread starts here
        bool done[CHUNK] = { false };              // indicate chunks completed
        for ( int i = 0; i < CHUNK ; i += 1 ) {    // for every chunk in the row
            try {
                _Enable {
                    if ( done[i] ) continue;       // chunk already completed ?
                    prework( chunks[i] );          // initial work
                    uEHM::poll();                  // NoUnblocking semantics
                    complete.work( chunks[i] );    // final work
                }
            } _CatchResume( Recall &r ) {
                chunks[r.i] = r.correction;        // replace erroneous data
                prework( chunks[r.i] );            // redo work
                complete.work( chunks[r.i] );
                done[i] = true;                    // mark work completed for this chunk
                if ( i == r.i )      _Throw;       // abort iteration
            } catch( Recall ) {}                   // iteration aborted ?
        }
    }
};

void uMain::main() {                               // program starts here
    uProcessor p[TASKS];                           // create kernel thread per worker
    int space[TASKS][CHUNK];                       // space for each worker
    Worker *w[TASKS];                              // workers
    for ( int i = 0; i < TASKS; i += 1 )
        w[i] = new Worker( space[i] );             // create worker tasks and start running
    for ( int i = 0; i < TASKS * CHUNK; i += 4 ) { // fix every 4th chunk across all tasks
        _Timeout( uDuration( 0, 100 ms ) );        // delay before next recall
        _Resume Recall( i % CHUNK, 3 ) _At *w[i / CHUNK];
    }
    for ( int i = 0; i < TASKS; i += 1 )
        delete w[i];                               // join with workers when finished
}
```

Figure 4.7: Correct Faulty Data with Resumptions

As before, removing the line with the "NoUnblocking semantics" comment produces a version of the program for use with the new unblocking semantics, whereas the line remains when there are no unblocking semantics. The NoUnblocking version has an explicit poll to ensure that if an exception is delivered during prework, it is detected before entering the monitor. Even with this advantage, using the new semantics yields a run-time of between 10.85 s and 10.88 s compared to between 14.41 and 14.43 s without it (ten runs each). This difference is explained by pseudo-blocking. With the new semantics, tasks that are lined up to enter the monitor and receive a resumption to fix their data can step out and redo the prework on the new data while still being conceptually blocked on monitor entry (and without losing their position in the queue). Without the new semantics, a task cannot react to the resumption until it gains ownership of the highly-contested monitor. Indeed, the difference of around 3.55 s is approaching the theoretical maximum of 5 s (20×0.25 s of prework). Hence, the ability to run while conceptually blocked, which could be called cheating, combined with the existence of a highly-contested resource results in a substantial performance increase. In general, pseudo-blocking can be exploited in a fashion similar to the one above in order to implement event-based programming or worker-thread pools.

4.6.3 Non-traditional Applications

Asynchronous exceptions with unblocking semantics allow for a very general form of asynchronous transfer of control. This control-flow mechanism has uses beyond traditional exception-handling as the following examples demonstrate. As always, it is up to the user to decide whether the complexity and/or the overhead of using exceptional termination[11] (stack unwinding, *etc.*) for non-exceptional purposes is justified for the additional capabilities gained.

Exploiting Explicit Task-Specific Wake-Up

The *Sleeping Barber problem* [Dij65, §4.2] consists of a barber (task) providing haircuts to customers (tasks) (see Figure 4.8). If the barber is busy with a customer, an arriving new customer sits down (waits) in the waiting room if there are any seats left, or balks (leaves) otherwise. If there are no customers, the barber goes to sleep (waits) until a customer arrives and wakes it.

In this particular variation of the sleeping-barber problem, the following additional properties are required. The barber must service waiting customers in the order they arrive, *i.e.*, in the order

[11] Resumption in μC++ is a light-weight mechanism compared to termination.

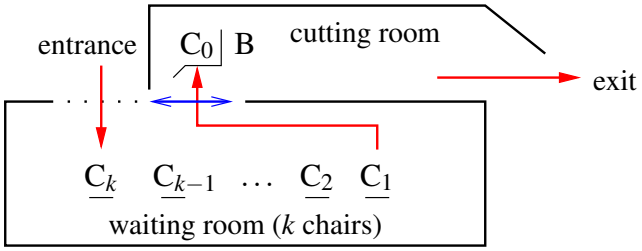

Figure 4.8: Sleeping Barber

they check the number of occupants in the waiting room. If no task is located in or examining the waiting room, then an arriving task must be able to examine the waiting room without blocking. A customer must block at most $O(1)$ times. The barber must block at most $O(n)$ times, where n is the number of customers. No busy-waiting is allowed in the algorithm itself.[12]

Employing different forms of synchronization, there are various solutions to this problem. It can be an interesting challenge to restrict the use of certain synchronization instruments, either for teaching purposes, or because many languages only support a small set of synchronization instruments. If only simple _Accept synchronization is employed in the sleeping barber solution (*e.g.*, to simulate Ada without requeue), the use of unblocking semantics can allow for a more efficient solution for the following reasons. After checking that there is enough space in the waiting room, a customer may need to wait for the barber. With only _Accept synchronization, it either has to block inside the monitor or outside on a monitor entry. The only way to block inside is to accept something, such as arriving customers or the barber. If it accepts customers, it may have to wake and block more than a constant number of times. If it does not accept them, arriving customers cannot check if the waiting room is full and immediately balk. Therefore, a customer that acquires the monitor and determines it must wait cannot wait inside the monitor, so it must leave and attempt re-entry. However, it is impossible to atomically leave a monitor and (re-)block on an entry queue without using additional synchronization facilities, like Ada's requeue. As a result, as soon as a customer leaves, another may enter the waiting room, decide to wait, leave, and block ahead of the original one, perturbing the arrival order. Without conditional entry (*e.g.*, in SR [AOC+88] or Concurrent C [GR89]), the barber is forced to accept customers in the order they block on the entry, *not* the order in which they arrive at the waiting room. The way to solve this problem is to ensure these two orderings are identical. Since there is no atomicity between

[12]Some spinning may be required in the implementation of higher-level mutual exclusion or synchronization instruments.

leaving a monitor and blocking on entry, the only alternative is to have at most one customer block on an entry queue. This approach requires the use of at least $\Omega(k)$ _Accept/entry queues (similar to the Ada *entry-family* mechanism), where k is the number of chairs in the waiting room, and customers and barber choose an entry queue they block-on/accept by arrival order, similar to ticket-based approaches. If the number of chairs is large or dynamic, this approach is impractical.

Exceptional unblocking can solve this problem using $O(1)$ entry queues, which the example in Figure 4.9 demonstrates. Note, the barber calls BarberShop::startCut to attempt to select the next customer from the waiting room. When the call returns, it means there is a customer in the barber's chair, and the haircut can be performed (not shown). Once the haircut is done, the barber calls BarberShop::endCut, which releases the customer just serviced. Customers call BarberShop::hairCut. After the call returns, either the customer balks (if the waiting room is full), or the customer received a haircut. Customers waiting for a haircut block on the synchronization queue of WaitingChairs::dummy. They are woken by an exception raised at them by the barber in BarberShop::getNext.

When an asynchronous exception is raised at a task blocked on any monitor queue, it is unblocked regardless of its position in the queue. Hence, the lack of atomicity between leaving and entry-blocking becomes irrelevant as the barber can unblock customer tasks from any queue in an arbitrary order simply by raising asynchronous exceptions at them. Exceptional unblocking thus serves as a replacement for conditional entry. All that the barber needs to know is the order in which the customers arrive, which the customers record as they arrive.

In general, the raising task need not own the instrument on which the propagating tasks are blocked, nor does it need to know where exactly they are blocked, which could be useful in a variety of similar problems. While FIFO servicing of synchronization/mutual exclusion queues is useful for ensuring fairness and predictability, at the same time, it can also impose an unwanted ordering for a particular problem, which requires additional code complexity to overcome. The ability to unblock tasks from a queue in an arbitrary order using asynchronous exceptions can simplify such code; thus, asynchronous exceptions are useful as a powerful control-flow tool even without an apparent exceptional situation.

```
Customer *customers[20];                                    // customers indexed by id

class BarberShop {
    _Event Wake {};
    _Monitor WaitingRoom {
        static const int maxchairs = 10;
        int count, next, chairs[maxchairs];
      public:
        WaitingRoom() : count( 0 ), next( 0 ) {}
        bool isSpace( int id ) {
          if ( count == maxchairs ) return false;           // all chairs in use
            chairs[ (next + count) % maxchairs ] = id;      // sit at end of the queue
            count += 1;                                     // number of chairs in use
            return true;
        }
        int getNext() {
            if ( count == 0 ) _Accept( isSpace );           // if waiting room empty, wait
            int id = chairs[next];                          // next customer for hair cut
            count -= 1;                                     // number of chairs in use
            next = (next + 1) % maxchairs;                  // new front of the queue
            _Throw Wake() _At *customers[id];               // unblock customer from ready/entry list
            return id;
        }
    } waitingroom;

    _Monitor WaitingChairs {                                // explicit waiting queue
      public:
        void dummy() {}                                     // never called
        void waitTurn() { _Accept( dummy ); }               // customers wait on ready/entry list
    } waitingChairs;

    _Monitor BarberChair {                                  // synchronizes two tasks in arbitrary order
        bool ready;                                         //   through member routine sync,
      public:                                               //   like a barrier with two participants
        BarberChair() : ready( false ) {}
        void sync() { ready = ! ready;   if ( ready ) _Accept( sync ); }
    } barberChair;

  public:
    void hairCut( int id ) {                                // called by customer
      if ( ! waitingroom.isSpace( id ) ) return;            // no space ? balk
        try { _Enable { waitingChairs.waitTurn(); }         // queue on waiting chairs and/or propagate
        } catch( Wake ) {
            barberChair.sync();                             // block for barber to finish previous cut
                                                            //   or unblock sleeping barber
            barberChair.sync();                             // synchronize with barber, complete my hair cut
        }
    }
    int startCut() {                                        // called by barber
        int id = waitingroom.getNext();                     // get id of next customer
        barberChair.sync();                                 // block for customer or unblock customer
        return id;
    }
    void endCut() {                                         // called by barber
        barberChair.sync();                                 // block for customer or unblock customer
    }
};
```

Figure 4.9: Sleeping Barber with Accept

Simulating Message Passing

The message-passing paradigm [Wal72] allows tasks to communicate and synchronize using messages rather than remote procedure-call. While μC++ does not support message passing directly, it is possible to construct it in a number of ways, including using asynchronous exceptions with unblocking semantics. The basic primitives of message passing are send and receive-any/receive-specific. Figure 4.10 shows a receiving task obtaining messages (exceptions of type Message) from a number of sending tasks (receive-any). _Enable < Message > allows the propagation of exceptions of type Message, while blocking on condition cv ensures the receiver blocks if no such message is available; unblocking semantics ensure the receiver is unblocked when a message is delivered.

The mechanism to support receive-specific is depicted in Figure 4.11. Its implementation relies on *bound exception matching* as it applies to asynchronously raised exceptions, *i.e.*, using the raising execution as a matching criteria for determining the most specific handler [BK06, §5.2],[13] and requires incorporating this concept into asynchronous propagation control. Extending the bound-execution matching to asynchronous propagation-control allows the latter to distinguish between asynchronous exceptions raised by different raising executions. The bound-execution matching of the _Enable ensures that only messages from the designated sender (senders[j] in this case) are eligible for propagation; the remaining mechanism ensures the receiver blocks and unblocks appropriately.

4.7 Summary

Allowing asynchronous exceptions to unblock a propagating task follows naturally from the desire to ensure timely handling of an exception, as well as from the abort characteristics of exceptional termination semantics. As demonstrated, such a feature can be implemented without the need for additional syntax or unusual programming techniques. As well, especially with elaborate synchronization protocols, this language feature can allow a programmer to write simpler, more intuitive code. In addition, when there is strong contention for a shared resource, pseudo-blocking can be used to increase concurrency, and thus, program performance. Furthermore, exceptional unblocking allows tasks to be pulled off blocking queues in an arbitrary order, which can simplify and optimize concurrent algorithms. Finally, exceptional unblocking (in combination with

[13] The reference calls asynchronous exceptions *non-local exceptions*.

```
_Event Message {                                    // exception carrying message
  public:
    string s;
    Message( string s ) : s( s ) {}
};
const int Senders = 10, Messages = 10;

_Task Receiver {
    uCondition cv;                                  // used to block Receiver until message arrives
    void main() {
        for ( int i = 0; i < Senders * Messages; i += 1 ) {
            try {
                _Enable < Message > {               // propagate exceptions of type Message
                    cv.wait();                      // wait for send
                }
            } catch( Message &msg ) {
                // process message from sender
            }
        }
    }
} receiver;                                         // create and start receiver

_Task Sender {
    void main() {
        string msg;
        for ( int i = 0; i < Messages; i += 1 ) {
            // generate message in "msg"
            _Throw Message( msg ) _At receiver; // send message
        }
    }
} senders[Senders];                                 // create and start senders

void uMain::main() {}
```

Figure 4.10: Send/Receive-any with Asynchronous Unblocking Exceptions

```
try {
    _Enable < senders[j].Message > {        // only allow propagation of Message-type exceptions
        cv.wait();                          //   raised by specific sender "senders[j]"
    }
} catch( Msg m ) {
    // process message from sender
}
```

Figure 4.11: Send/Receive-specific using Propagation Control with Bound-Execution Matching

bound-execution propagation control) can be used to emulate other concurrency mechanisms such as message passing in a simple and intuitive fashion. The variety of applications of this feature suggests there may be many more practical uses.

Implementing exceptional unblocking is complex, and programmers must be aware that some aspects of blocking instruments cannot be preserved when tasks are unblocked at non-deterministic times. Nevertheless, with careful use, exceptional unblocking provides a powerful mechanism for developing complex concurrent programs whose behaviour follows intuitively from normal exception-handling.

Chapter 5

Improving the Usability of Asynchronous Exceptions

As useful as structured exception handling is, it is also complex. This complexity makes it difficult to intuitively employ correctly. Especially novice programmers seem to struggle with the complexity of the exception-handling concept and its effect on control flow [SGH10]. Even experienced programmers misunderstand or misuse certain exception-handling features, *e.g.*, out of convenience [Eck07]. Asynchronous exception handling is even more complex than the traditional, synchronous kind. For this reason, the design of more advanced exception-handling features is complicated when taking usability aspects into account: It is tempting to add more advanced capabilities, and thus, complexity, to an already complex topic, making it too difficult to use in practice by the 'average' programmer. The unblocking semantics introduced in Chapter 4 are a good example of additional capabilities that can actually reduce conceptual complexity by allowing for more intuitive program-behaviour. In contrast, the full-asynchrony semantics described in Chapter 2 and implemented in Chapter 3 add capabilities to the already complex concept of asynchronous exceptions. Recall that previous work attempted to reduce this complexity by restricting asynchrony (see Section 1.5, p. 12). When full asynchrony is added, however safe its design is, the resulting non-determinism inevitably increases the overall complexity.

This chapter attempts to contain the complexity of asynchronous exceptions by presenting language features that assist the programmer in using them correctly. Many of the concepts also apply to synchronous exception-handling, and thus, can aid in the understanding of exception handling in general.

5.1 Usability Challenges

Shah *et al.* make the observation that inexperienced programmers tend to ignore exception handling beyond what is required of them by the language[1], either because they do not understand the mechanism entirely or do not want to deal with it until their program breaks due to an exception [SGH10]. They are mainly interested in the "*happy path*" of the program, *i.e.*, not the *exceptional path*, in which exceptions affect control flow. The following discussion identifies the aspects of exception handling that lead to their misuse. These *usability challenges* are not problems of the underlying exception-handling concepts, but instead problems of understanding them or using them properly. Once these challenges are understood, it is easier to construct solutions to help overcome them.

5.1.1 Rarity

By definition, exceptions, or rather, exceptional raises/propagation, are rare, which can lead to two problems. First, from a work-economy point of view, it may appear sensible to spend more time on the main path of control (happy path), rather than dealing with exceptional control-flow that rarely affects the program (also see Section 1.5, p. 13). Second, since exceptions occur rarely, the exceptional path is not tested as thoroughly as the happy path. Cui and Gannon claim that exception handling is the least tested part of a programming interface [CG92]. It is a reality that most programmers do not anticipate all possible circumstances and boundary conditions in which their program is expected to perform correctly. They test their programs, and when one of these conditions occurs and causes the program to behave incorrectly, they fix the program accordingly. Since exceptions are rare, such conditions in connection with exception handling occur very rarely and are easily missed during testing, meaning these cases are easily overlooked, and thus, the program fails when they are encountered.

5.1.2 Complexity

Exception handling involves complex control-flow. First, since handler association is dynamic, given a raise or a propagation point, the corresponding handler site cannot be found statically in general. Second, since the transfer of control can be non-local, the handler site and

[1] The main language of study, Java, through its checked exception mechanism, forces a programmer to acknowledge the possibility of exceptional propagation, though it is questionable whether this activity leads to a true understanding of a program's exception handling.

raise/propagation point can be in separate routine scopes or even translation units. Third, with termination semantics, the abortive semantics of exception propagation with its associated stack unwinding is destructive and unintuitive. For example, a sequence of statements is not guaranteed to be executed as depicted, but can be aborted at any point an exception can be propagated, which is generally difficult to anticipate and comprehend.

5.1.3 Strictness

Exception handling is strict in the sense that it cannot be ignored passively; it is an active phenomenon [BHM02]. If an exception is raised/propagated, it affects the control-flow of the program, especially with terminating semantics. Failure to recognize a potential propagation point often has catastrophic consequences for the program. Hence, it becomes tempting to over-handle exceptions, *i.e.*, employ blanket handlers covering as many exception types as possible that do nothing useful, which weakens the usefulness of exception handling.

5.1.4 Asynchrony

Asynchrony exacerbates both control-flow complexity and rarity of exception handling. Obviously, asynchrony increases the difficulty of determining whether a sequence of code is abortable or is executed to completion (see Section 1.4, p. 10). Additional factors like asynchronous propagation-control and the asynchrony model add to the complexity, making asynchronous exception handling more difficult to understand. In a sense, exceptions also become rarer due to asynchrony: The probability of a rare exceptional propagation is distributed over many possible statements (over which asynchronous propagation is enabled). As a result, more locations can propagate an exception, each with a lower probability, which makes it even harder to test a program with regard to propagation of an exception at a given location.

5.1.5 Analysis

The preceding challenges have two aspects: mind-set issues (as touched upon in Section 1.5, p. 13), and fundamental challenges (stemming from the fundamental nature of exception handling). This thesis does not try to address the mind-set issue. Attempting to force exceptions into the programmer's consciousness by requiring the explicit handling or propagation of checked exceptions [GJSB00] has not been popular, and is considered a failure by some [Hei03, Eck07].

Trying to address the fundamental aspects of these challenges is difficult since what makes (asynchronous) exception handling challenging is also what makes it powerful and useful, *e.g.*, recall Chapter 2, which examines tools to control and manage asynchrony and their inherent compromises between capabilities and safety. In particular, the strictness of exceptional propagation cannot be weakened without weakening the entire concept. Its control-flow complexity is the flip-side of the immensely powerful control-flow possibilities of exception handling. Weakening it, *e.g.*, by employing sequels [Knu87], means weakening its capabilities. Rarity and asynchrony-induced rarity show their greatest effect in the sparse testing they allow. The way to overcome this issue is to artificially increase the frequency of exceptional propagation during testing. Finally, a promising vector for attacking the associated usability-challenges is to provide additional run-time information that documents what goes on in the exception-handling mechanism of a program; instead of reducing the complexity of the control flow, additional information is provided in order to make it easier to comprehend. The remainder of this chapter presents features to address some of these usability issues.

In particular, in order to address the rarity of exceptional propagation and the resulting lack of testing, *active exception assertions* are introduced, which increase the frequency of propagation during testing. In order to address the complexity of exception handling, features providing additional run-time information are introduced aimed at giving the programmer more insight about the exception handling in a program: *Passive exception assertions* can be used to formulate assumptions about synchronous and asynchronous exception propagations. The ability to generate stack traces is added to provide information about the call stack at the time of synchronous or asynchronous propagation. In order to provide additional information particular to asynchronous exception handling, features are provided to query and display the asynchronous propagation-control stack. Finally, a mechanism is introduced to log and query all events that influence asynchronous exception handling, thus providing a comprehensive tool for its debugging and testing.

5.2 Exception Assertions

The frequency of exception handling can be increased by artificial propagation of exceptions in an area to be tested, meaning exceptions need to be generated specifically for testing purposes. It remains to determine how and under what circumstances such exceptions are to be propagated. Clearly, just injecting an exception into a block of code, regardless of the state of the program, is

insufficient. An exception is a representation of an exceptional situation. Without an exceptional situation and its associated state, an exception is meaningless. Conversely, a program cannot be faulted for not handling an exception in situations in which it cannot occur. Note, handling here means more than just having a handler in place to deal with the control-flow effect of the exceptional propagation; it is relatively easy to make sure a handler is in place, but it is difficult to ensure the handler deals with the exceptional situation correctly, *e.g.*, by releasing resources. Hence, solutions like in [CM08], which unconditionally inject exceptions into a program, are too crude to be useful as the program state that can lead to an exceptional propagation is not considered. Tracey *et al.* try to address this problem by generating input data to trigger built-in constraint exceptions [TCMM00], but their approach, aimed at program verification, is limited to the SPARK-Ada dialect and its built-in run-time exceptions. The conditions under which these are raised are well-defined as part of the language specification, and are thus relatively easy to analyze. This solution does not address user-defined or library-defined exceptions, which are far more plentiful in most modern languages. The conditions under which such user exceptions occur are difficult to determine, *e.g.*, by programatically analyzing source code, especially when not all source code is available. Programmers, on the other hand, can make assumptions about when these conditions are met, and the correctness of these assumptions is a good way to test the programmers' understanding of the exception handling in their programs.

5.2.1 Inject under Condition

As a compromise, the following scheme is proposed: For testing purposes, exceptions are injected into blocks of code when a user-specified condition is met. In order to test asynchronous exceptions, the programmer supplies, as part of the _Enable statement, an *exception assertion*, which contains the instantiation of the exception object, as well as the condition to be tested, *e.g.*,

```
_Enable < PacketLoss(TCP, id) ? ( tcp_window_size > 0.8*max_tcp_window_size ) > {
  ...
}
```

Here, asynchronous propagation of PacketLoss is expected to occur when the condition tcp_window_size > 0.8*max_tcp_window_size is met. When the compiler is set to a special testing mode, then, as soon as the condition is met, an exception is instantiated with the parameters (TCP, id) and propagated from within the _Enable block. Naturally, the identifiers TCP, id, tcp_window_size, and max_tcp_window_size must be visible by the _Enable statement, *i.e.*, visibility *inside* its block is insufficient.

Note, this mechanism is intended to allow for the increased testing of exceptional situations *given* that the condition is met. The condition itself may or may not be met frequently during a program's life time, and may be difficult to force artificially. Ultimately, it is up to the programmer to specify a condition that allows for adequate testing while reflecting realistic exceptional situations.

The condition mechanism provides the programmer with a very general way to trigger the exception assertion, *i.e.*, propagate the exception. Hence, the following two extensions to the mechanism could be implemented by clever use of an appropriate condition, but are important enough to warrant providing them as explicit features.

5.2.2 Inject with Probability

With terminating semantics, it may not be desirable to always trigger the exception assertion once the condition is met, *e.g.*, the programmer may want control flow to potentially proceed into the _Enable block. A probabilistic measure can therefore be added to the mechanism that causes propagation to occur with a specified probability, and thus, allows for the coverage of a greater number of test cases over different runs of the program. This refined control is achieved by supplying an *individual probability* as part of the exception assertion, *e.g.*,

```
_Enable < PacketLoss(TCP, id) ? ( tcp_window_size > 0.8*max_tcp_window_size ), 0.35 > {
  ...
}
```

augments the previous exception assertion such that it is triggered only 35% of the time when the assertion condition is fulfilled.

5.2.3 Grouping

During testing, it may be useful to group certain exception assertions and assign probabilities to entire groups. One reason is to test each group separately, *e.g.*, by assigning a probability of 0 to all other groups. To achieve this capability, it is possible to associate an integer ID with an exception assertion, *e.g.*,

```
_Enable < PacketLoss(TCP, id) ? ( tcp_window_size > 0.8*max_tcp_window_size ), 0.35, 7 > {
  ...
}
```

which assigns the ID 7 to the sample exception-assertion.

To allow for maximum flexibility with regard to determining group and individual probabilities, and to enable dynamic changes of these parameters during testing, the following routine is

available:

```
double getAssertProbability( double probability, int group )
```

The system queries it whenever a trigger point is reached by supplying the individual probability and group ID, and receiving the dynamically calculated probability for the trigger point as a result. A default implementation is supplied that simply returns the individual probability specified. Redefining this function in a program causes the linker to choose it over the default implementation. Thus, a programmer can realize any desired probability scheme, *e.g.*,

```
double getAssertProbability( double probability, int group ) {
    static uOwnerLock l;
    static int counter = 0;

    l.acquire();                                    // protect counter from concurrent access
    int count = counter += 1;                       // store current counter
    l.release();
    switch ( group ) {
      case 1:
      case 2:
        return probability*(1 - 10.0/count);        // note: negative probabilities are
      case 3:                                       //  converted to 0.0
        return 0.0;
      default:
        return probability;
  }
}
```

In the example above, the probability of assertions with IDs 1 and 2 is slowly ramped up to their specified value, *e.g.*, to let the program settle down before assertions start to trigger. Those with ID 3 have their probability set to zero, *i.e.*, triggering of these assertions is turned off. Any other exception assertion is triggered according to its supplied individual probability.

5.2.4 Distribution

The location inside the _Enable block where the triggered exception is injected (propagated) is important. The simplest implementation triggers it when the block is first entered. However, while this solution is still a useful test of this likely propagation point (since _Enable-block entry is a poll point), it is equally important to cover the entirety of the _Enable block due to the asynchronous nature of possible propagations within it. A better scheme simulates the asynchronous nature of the _Enable block by distributing different sub-trigger points (in effect, separate exception assertions of the same condition and ID) among statements throughout the _Enable block. In this case, the probability needs to be divided among these different sub-trigger points, but an even division may not be desirable: Statistically speaking, the earlier trigger points are likely to be encountered more often than the later ones with terminating semantics because exceptional

propagation precludes the testing of later sub-trigger points. Since exception injection is meant to test for unhandled exceptions, and unhandled exceptions ultimately manifest themselves (and cause the greatest damage) using terminating semantics, it makes sense to assume this form of propagation. It is therefore better to skew the division of the individual (conditional) probabilities p_i in a fashion that favours later points such that if there are 0..n-1 sub-trigger points numbered by lexical occurrence, they each have the same overall (unconditional) probability P/n if P is the probability assigned by the programmer to the overall exception assertion. In other words, for a sufficiently large number of program runs, all sub-trigger points should trigger the same number of times. The conditional probability p_i of the i^{th} sub-trigger point, *i.e.*, the probability that this sub-trigger point triggers under the condition that it is reached (none of the earlier ones trigger), is therefore determined as follows:

$$\begin{aligned} & \frac{P}{n} &=& \left(1 - P\frac{i}{n}\right) p_i \\ \Leftrightarrow \qquad & p_i &=& \frac{P}{n - iP} \end{aligned}$$

Of course, since control flow inside the _Enable block can be arbitrary, there is no guarantee that execution actually progresses lexically from top to the bottom. Without extensive control-flow analysis, however, this heuristic seems reasonable.

5.2.5 Extending to Other Statements

While the asynchronous nature of the _Enable block presents special challenges to understanding and testing exception handling, exception assertions are not restricted to just the asynchronous domain. It may be useful to test synchronous exception handling in this fashion as well. Hence, the concept is extended by using a special try-statement, *e.g.*,

```
try < PacketLoss(TCP, id) ? ( tcp_window_size > 0.8*max_tcp_window_size ), 0.35, 23 > ;
```

Here, the exception is triggered inside the try-statement. A try-statement is used for ease of parsing; it need not have any guarding handlers. Unlike with the _Enable block, in which the exception assertion has the additional semantics of enabling propagation of the exception, the exception assertion as part of the try-statement has no additional semantics beyond exception-assertion testing. It may not be useful to use this mechanism to inject exceptions into existing try-blocks that guard against synchronous propagations, *i.e.*, by augmenting existing try-blocks with the exception-assertion syntax above; the distribution of triggering points across the block

makes little sense in this case. However, augmenting an existing try-block with an assertion condition can be useful for the passive assertion check introduced in the following section.

5.2.6 Passive Exception Assertion

Assertions are useful in challenging and verifying a programmer's assumptions about a program. If an assertion fails, a significant run-time effect is produced, *e.g.*, terminating the program. Exception assertions can be used in a similar fashion, from which they derive their name. The injecting of exceptions discussed so far is called *active assertion*. Using the same exception assertion syntax, an alternative mode[2] is *passive assertion*, in which an exception assertion only performs an action when a 'real' exception of the same or derived type is propagated out of the block it guards: If the assertion condition is not met, the programmer is warned.

In the previous example, if a PacketLoss exception is propagated out of the try-block and the assertion condition is *not* met, the run-time warns the programmer by printing the following message to *standard error*:

```
Assertion (tcp_window_size > 0.8*max_tcp_window_size) failed for raised
exception of type PacketLoss.
```

In this way, the programmer is warned of a misleading assumption about the exception handling inside the program, and can take measures to rectify the issue, as well as gain a better understanding of the program.

When not debugging, *i.e.*, in the release binary, all testing (passive or active) semantics of exception assertions are removed, so program performance is unaffected by the testing code.

5.2.7 Example

The example in Figure 5.1 shows resumption being used in event-driven programming. A central routine eventLoop collects events via GotInput resumptions raised at its execution from an external task, and distributes them to an array of N processing servers. When a server is finished with its calculation, it sends the result via a FinishedCalc resumption back to the execution executing eventLoop, which outputs the result. The variable outstanding keeps track of how many calculations are underway.

[2]Active assertions are used in compilation units compiled with the -assert flag. Passive assertions are implicitly enabled using the -debug flag.

```
const int N = 20;

_Event GotInput {
  public:
    int input;
    GotInput( int i ) : input( i ) {}
};
_Event FinishedCalc {
  public:
    void *result;
};
_Event End {};

void eventLoop() {
    int outstanding = 0;
    try {
        for (;;) {
            _Enable < GotInput(rand() % N) ? (outstanding < N), .75 >
                    < FinishedCalc ? (outstanding > 0), .25 > {
                std::cout << outstanding << "  ";
            }
        }
    } _CatchResume ( GotInput &gi ) ( outstanding ) {
        if ( gi.input == 0 ) _Throw End();          // some termination condition
        outstanding++;                              // events being processed
        server[rand() % N].process( gi.input );     // pass event to server
    } _CatchResume ( FinishedCalc &fc ) ( outstanding ) {
        outstanding--;                              // event processed
        outputResult( fc.result );                  // output the result
    } catch ( End ) {                               // catch reraised exception
        std::cout << std::endl;                     // finish up
    }
}
```

Figure 5.1: Exception assertion example

In this example, an active exception assertion implicitly injects exceptions that would otherwise be raised by the external task. It can thus be used to test this part of the code when no external task is present. The line

```
_Enable < GotInput( rand() % N ) ? (outstanding < N), .75 >
        < FinishedCalc ? (outstanding > 0), .25 >
```

does two things: First, it enables propagation of GotInput and FinishedCalc exceptions. Second, it makes the assertion that GotInput exceptions are only propagated when outstanding < N, while FinishedCalc exceptions are only propagated when outstanding > 0. These conditions seem natural since there are only N servers to process requests and there can be no results received when there are no outstanding calculations. If active assertions are turned on, GotInput exceptions[3] are to be injected with a probability of 75%, while FinishedCalc exceptions are injected with 25% probability. The reason for this choice may be that inputs are expected to occur three times faster than calculations. In order to observe what happens over the program's run-time, the value of

[3]They are instantiated with rand() % N as an argument: Eventually, the argument is 0, and the loop terminates.

outstanding is printed out periodically. Here is one of the outputs generated by the program:

```
0 1 3 4 6 6 7
```

It is clear that the value of outstanding tends to rise, which is explained by the number of GotInput exceptions exceeding that for FinishedCalc, until the terminating condition is met. When equal probabilities are used as in

```
_Enable < GotInput( rand() % N )  ? (outstanding < N), .75 >
        < FinishedCalc ? (outstanding > 0), .75 >
```

it is not surprising that the resulting output hovers in the range of 0-1:

```
0 1 1 1 1 0 0 1
```

When active assertion checking is turned off, *i.e.*, only passive checking remains, and the code is tested by generating a lot of GotInput events, here is what the output potentially looks like:

```
0 2 2 2 3 5 6 8 ... 19
Warning:  On _Enable block entry: Assertion ( outstanding < N )
  false for raised resumption of type GotInput
```

The nice thing about this message is that it alerts the programmer to a misconception in the program design. When there are more than 20 outstanding calculations, a server needs to perform more than one at a time, which it may not be designed to do. At the very least, this message gives the programmer an idea under what circumstances such boundary cases are reached.

5.2.8 Implementation

Exception assertions are implemented in μC++ largely by performing code transformations via the μC++ translator.

Active Assertions

Implementing active assertions is relatively straight-forward. It requires injecting code sequences into user code that evaluate the supplied condition and probability, and raise an exception depending on the result, *e.g.*,

```
if ( ( b > a ) && triggerAssertion( 0.5 , 2 ) )
    _Resume(...);
```

where triggerAssertion is a routine that, given an individual probability and assertion ID, returns (by calling getAssertProbability) whether a propagation should be triggered. The triggered exception is raised using resumption semantics as the exception-assertion syntax does not specify

whether terminating or resumption semantics are required. Since the default resumption handler reraises the exception using terminating semantics, both semantics are covered. Note, this method does not permit testing terminating semantics where a resumption handler exists for the same or derived exceptions as the resumption handler prevents the exception from being reraised with terminating semantics, *e.g.*,

```
try {
    _Enable < PacketLoss(TCP, id) ? (true) > {
        ...
    }
} _CatchResume( PacketLoss ) { ... }          // handles all injected PacketLoss exceptions
} catch( PacketLoss ) { ... }                 // never executed
```

Here, if there is a terminating raise (throw) within the _Enable block that is supposed to be caught by the catch-handler, this exception path cannot be tested by this active assertion. Any injected PacketLoss exceptions are handled by the resumption handler since they are raised with resumption semantics. The default resumption handler, which would otherwise reraise the exception with terminating semantics, is never executed as the supplied resumption handler takes precedence, so the injected PacketLoss exceptions are never raised with terminating semantics. For asynchronous exceptions, this restriction is no drawback since asynchronous exceptions cannot be raised with terminating semantics anyway (only the absence of a resumption handler turns a raised asynchronous exception into one with terminating semantics, see Section A.3, p. 161). For synchronous exceptions, it can make sense to have both terminating and resumption handlers for the same exception type and guarded block. However, such code makes most sense when a resumption handler reraises the exception with terminating semantics, in which case the injected exception is reraised as well, meaning the test case covers the intended functionality. Other cases of a resumption and termination handler of the same type guarding the same block should be rare.

Distributing the triggering checks over an _Enable block requires inserting these checks between statements within the block, where care must be taken that the sub-triggers have the correct individual (conditional) probabilities p_i such that they combine to the total probability associated with this assertion and ID (see Section 5.2.4, p. 131).

Note, when simply copying the distributed triggering checks as above, evaluating the condition can be problematic since the binding of the identifiers at the location of the trigger check may not be the same as the one at the entry of the enable block, *e.g.*, the code snippet

```
int a = 10;
int b = 0;
_Enable < PacketLoss ? ( b > a ), ... > {          // what exactly is enabled
    ...                                            //   is irrelevant here
    int b = 100;                                   // new b variable hides old b
    ...
}
```

is transformed into

```
int a = 10;                                        // user code
int b = 0;                                         // user code
_Enable < PacketLoss > {                           // user code
    if ( ( b > a ) && triggerAssertion( ... ) )    // generated code:
        _Resume(...);                              // first sub-trigger
    ...
    int b = 100;                                   // user code

    if ( ( b > a ) && triggerAssertion( ... ) )    // generated code:
        _Resume(...);                              // second sub-trigger
    ...                                            // user code
}
```

Now, the condition check of the first sub-trigger is false while that of the second sub-trigger is true even though the condition is unchanged. What is changed is the binding of identifier b. This problem also affects parameters supplied for exception instantiation. This issue arises from the difficulty of capturing a closure in C++. The new C++0x standard solves this problem by providing lambda functions [ISO10]. For this thesis, and until the C++0x standard is finalized, the simple approach above suffices, but should be used with caution.

Passive Assertions

Since μC++ exceptions are dual [Mok97], they can be raised using terminating and resumption semantics[4]. For a given passive exception assertion, the semantics through which the exception is raised cannot be anticipated in general, so a passive exception assertion check needs to be implemented for both resumption and terminating semantics.

Implementing passive assertion checks for termination semantics (throw) is straight-forward, except for how and when the assertion condition is checked. One option is to evaluate the assertion condition when or just after the exception is raised since the condition is related to the exceptional situation, and thus, the raise. However, it is very likely the factors causing an exceptional raise are (in part) local to the raise site, and it is unlikely that an exception assertion located far from the raise site has access to local state. Otherwise there is close coupling between lower-level (raise site) and higher-level (exception assertion site) code, which is generally undesirable

[4] Asynchronous exceptions can only be raised with resumption semantics, but there is no reason to restrict exception assertions to asynchronous exceptions alone.

from a software-engineering perspective.

An alternative option, especially if the exception assertion is far from raise point, is for the assertion condition to query the state when the exception is propagated into or out of the block to which the assertion is attached.

From a technical point-of-view, both options are tricky. If the condition is to be evaluated at the raise, there is the problem that raising code or additional code that is executed as a consequence of the raise generally cannot see the scope in which the condition exists, and therefore, cannot evaluate it. Again, a closure can allow the execution of code at the assertion scope, which is a capability that the lambda function of the new C++0x standard provides. With this functionality, it is possible to capture the code that checks the assertion condition in a lambda, store this lambda, and then evaluate it when the raise occurs.

In this work, the alternative option is chosen, *i.e.*, the condition is evaluated when propagation exits the block to which the assertion is attached (for termination semantics). The trickiness of its implementation again lies in evaluating the condition in the correct scope. This approach is easy to implement for termination semantics as an exception handler can naturally see the assertion condition and evaluate it. The _Enable- or try-block is enclosed by another try-block that catches the exception, evaluates the assertion, issues a warning if necessary, and finally reraises the caught exception, *e.g.*, the previous example becomes

```
try {
    _Enable < PacketLoss > {                // _Enable block from previous example
        ...
    }
} catch ( PacketLoss ) {
    if ( ! ( b > a ) )                      // assertion failed
        /* issue warning */
    throw;                                  // reraise exception
}
```

Since the scope of the catch clause includes that which exists just before entering the _Enable block, the bindings of a and b are as expected. Note that cleanups (e.g., destructors) run between raise and catch can potentially alter the assertion condition, *i.e.*, its value can be different than at the time of the raise.

Resumption semantics are more complicated to implement. μC++ currently implements resumption handlers as routines with global scope, and these generally cannot see, and thus, cannot evaluate the condition-part of the exception assertion. Again, using lambda functions according to the new C++0x standard solves this problem as they provide a closure, and thus, allow a re-

sumption handler to evaluate the condition in the correct scope. Until the new standard comes into effect, the assertion condition for resumptions is evaluated when first entering the block to which the assertion is attached, as opposed to when the exception is propagated through it. For symmetry, this evaluation at block entry is also performed for terminating semantics (in addition to the evaluation performed after the raise as discussed above). Conceptually, the resulting code looks as follows:

```
{
    bool firstEval = ( b > a );
    try {
        ...                                              // could be a synchronous raise
    } _CatchResume ( SomeException ) {
        if ( ! firstEval )                               // assertion on block entry failed
            /* issue first warning;
               implicit return to detection/raise point due to resumption */
    } catch ( SomeException ) {
        if ( ! firstEval )                               // assertion on block entry failed
            /* issue first warning */
        if ( ! ( b > a ) )                               // assertion at propagation failed
            /* issue second warning */
        throw;
    }
}
```

Under the assumption that the condition does not change between entering the block and the propagation through it, this implementation detail is benign, but there is no guarantee that this assumption is correct. While the implementation is not ideal, any unexpected behaviour results from unusual coding style, and should therefore affect few programs. In comparison to having no mechanism for testing exceptions, the presented approach is a significant improvement. Once lambda functions make it into the language standard, the implementation can be altered to take advantage of them, solving the issue discussed above.

Appendix C, p. 171 shows a complete example of all code transformations performed in order to implement exception assertions.

5.3 Run-time Information

One of the most frustrating aspects of debugging exception-handling code is the fact that, at least with termination semantics, useful information vanishes during exception propagation as part of stack unwinding. Interestingly, in the case of exceptions that are not caught, information may remain available if there is no need to unwind the stack when the program is about to be terminated. The C++ standard leaves it undefined whether the stack is to be unwound in such a case [Int98, §15.3.9], and consequently, implementations such as *Sun WorkShop 6 update 1 C++ 5.2* and

GCC 4.3.3 do not unwind the stack. Hence, with such implementations, debuggers are able to inspect program state at the point-in-time of the exceptional raise/asynchronous propagation. If exceptions are caught, however, *e.g.*, to print meaningful error messages or for aggregation, the prior state of the program at the point of asynchronous propagation is lost. Since exceptions are rare, it is important to preserve as much information as possible since the exceptional situation may not be easily recreated. This section examines what information is the most useful and should be preserved and how this preservation is accomplished. Note, in general, such additional information should only be collected while debugging so as not to negatively affect the performance of the final release.

5.3.1 Stack Trace

The *stack trace*, *i.e.*, the listing of call frames on the stack, is one of the most useful pieces of information available about the state of a program. In general, it is helpful in determining the control-flow that led to a particular program state. In the domain of exception handling, it is especially useful as propagation and handler precedence follow the call stack. It is for this reason that languages like Java or those working on top of .NET expose a stack trace and attach it to a raised exception object [GJSB00, Micb]; for Java, printing the stack trace to the console is the default behaviour if an exception goes uncaught. For fully-asynchronous exceptions, the top-most stack frame is especially important as it contains the instruction address at which the exception was propagated, which cannot be anticipated.

Usage

Hence, there is good reason to store a stack trace inside the exception object and provide an interface to access it programatically, *e.g.*,

```
try {
    ...
} catch ( PacketLoss e ) {
    e.printStackTrace( std::cout );
}
```

which produces a trace of the call-stack like

```
0x8066280:    0x80661fe  work2()
0x80668b6:    0x80668ab  foo::work(int)
0x80668c9:    0x80668b8  foo::work3(int, int, int)
0x806639b:    0x806635b  fred(int)
0x8066375:    0x806635b  fred(int)
0x8066375:    0x806635b  fred(int)
0x8066437:    0x806639e  uMain::main()
```

in which the instruction address is displayed followed by the starting address and the name of the called function. When an exception is propagated out of a task's starting routine, the default termination handler prints a stack trace before the program is aborted.

Implementation

There are two basic ways of implementing a stack trace. The first is to walk the stack directly, which is fast but architecture-dependent and usually requires a frame pointer. The second method, and the one employed here, is to use the interface to the unwinder provided by _Unwind_Backtrace (see Section 3.6.4, p. 68). While invoking the unwinder is generally slower, it provides for a consistent interface over multiple architectures, and thus, future compatibility. As exceptions are rare in nature and a stack trace is only generated when an exception is propagated, the overall cost attributable to the process of generating stack traces should be relatively small.

5.3.2 Propagation Control Snapshot

Another factor that affects asynchronous exception handling is asynchronous propagation control. Especially with restricted asynchrony and the corresponding infinite scope of _Enable blocks, finding the responsible _Enable block is more complicated because the state of the propagation control inside of an execution is constructed dynamically along the call stack. This information is especially useful in cases where propagation is expected but does not occur (failure to propagate, see Section 3.5, p. 64). An interface for accessing the current state of propagation control can therefore be useful for debugging purposes. The routine uEHM::printFullEnableDisableState(std::ostream &) prints out a representation of the complete current asynchronous propagation-control stack, *e.g.*, the program in Figure 5.2 produces the following output:

```
-joe
+john::mary
-john
+mary, +fred, +john
+fred
-<All>
+<All>
-<All>
```

```
_Event fred{};
_Event mary{};
_Event john{ public: _Event mary{}; };
_Event joe{};

void foo () {
    _Enable < mary > < fred >< john > {
        _Disable < john > {
            _Enable < john::mary > {
                _Disable < joe >  {
                    uEHM::printFullEnableDisableState(std::cerr);
                }
            }
        }
    }
}
void uMain::main() {
    _Enable {
        _Disable {
            _Enable < fred > {
                foo();
            }
        }
    }
}
```

Figure 5.2: Example of printing state of asynchronous propagation control

where types preceded by a "−" represent a _Disable of that type and those preceded by a "+" represent an _Enable. To determine whether propagation of a particular type is enabled, it is necessary to examine the propagation-control stack from the top down until a directive is found that fits that type, *e.g.*, john, despite having both _Enable and _Disable directives on the stack, is disabled since the `-john` is higher up the stack.

To assist in evaluating the complete propagation-control stack, the following algorithm transforms it into canonical form:

- For each directive top-to-bottom until end-of-stack or an `<All>`-directive is found:
  - ∗ if type is disabled and not matched in *enabled-set*, insert it into *disabled-set*.
  - ∗ if type is enabled and not matched in *disabled-set*, insert it into *enabled-set*.
- if `+<All>` found, output *disabled-set* followed by `+<All>`.
- otherwise output *enabled-set* followed by `-<All>`.

A type T is matched in a set, if that set contains a type S such that catch (S) handles propagations of T, *e.g.*, T inherits publicly from S, or they are identical.

The routine uEHM::printEnableDisableState(std::ostream &) implements this algorithm, so by calling

```
uEHM::printEnableDisableState( std::cerr );
```

the propagation-control stack above is reduced to:

```
+john::mary
+fred
+mary
-<All>
```

This output indicates that there are three exception types whose propagation is enabled. Propagation of any other exception type is disabled.

Alternatively, it may be useful in certain situations to check whether the propagation of an exception is enabled and alter control flow dependently. The macro propagationEnabled(id), where id is a type-name or a variable-name, provides this functionality. This feature is especially important for library routines as they have no control over the code that calls them. An example use of this macro is:

```
void libraryRoutine() {
    if ( propagationEnabled( PacketLoss ) ) {
        exceptionSafeSubRoutine();
    } else {
        subRoutine();
    }
}
```

In this example, subRoutine is fast but not exception-safe with regard to asynchronous PacketLoss exceptions. Hence, it is used only if propagation of PacketLoss is disabled; otherwise, the slower but exception-safe alternative exceptionSafeSubRoutine is called.

5.3.3 Total Event Log

Even for experienced programmers, the proper use of asynchronous exceptions is challenging. The inherent non-determinism makes it difficult to reason about a program or recreate a specific situation affecting exception handling. The ability to go back in time and review all events that affect (asynchronous) propagation can therefore be a useful tool in a programmer's debugging arsenal. For this reason, the following important events related to (asynchronous) exception handling are recorded and stored in a central log, with the following data:

1. Raise (unique exception ID, type, source, target)
2. Detection (unique exception ID, detecting execution, execution state)
3. Propagation (unique exception ID, instruction address at propagation)
4. Entering an _Enable / _Disable block (execution, block start/end, types controlled)
5. Exiting an _Enable / _Disable block (execution)

Collecting and retrieving the log information does not affect the program beyond potential lock contention, *i.e.*, the process of collecting and retrieving the information is transparent, and does not affect the behaviour of the rest of the program beyond slowing it down and influencing scheduling decisions (which are unpredictable in any case). The log can be accessed an arbitrary number of times and at any time during program execution, including from within a debugger if the debugger supports the execution of arbitrary routines (see definitions in Appendix Section D, p. 175).

In order to collect less information, the routine uDefaultEventLogMode() can be defined to specify which events are logged, *e.g.*,

```
int uDefaultEventLogMode() {
    return EHMdebug::prop | EHMdebug::detect;
}
```

only collects information about propagations and detections. The collection of information can be turned off by returning 0, which is the default.

Examples

Figure 5.3 shows an example of how this log information might be used. It defines a data structure and a routine to access the log and calculate statistics about the asynchronous exception queues of all executions inside a program. When the routine is run, it iterates through the log analyzing it. If a raise is encountered, it sets up a mapping between an exception ID and its target execution (*i.e.*, where it propagates). Then the data structure associated with this execution is incremented, meaning the length of its exception queue has increased by one. If a propagation is encountered, the data structure associated with the execution is retrieved and decremented, meaning the length of its exception queue has decreased by one. Since the data structure records a running sum of the queue length for each event (raise and propagation), an average can be calculated at the end by dividing this sum by the number of recordings. In addition, the data structure also takes sample recordings of the queue length on every step'th event. Using the print method, it can then print out a graph of the queue length over time. Figure 5.4, p. 146 shows part of the output for running averageQueue at the end of a modified version of EHM4.cc (a standard test program that is part of the μC++ distribution). Here, the queue length of execution 0x9c0b378 fluctuates with an average of around 300, whereas the queue of execution 0x9c13778 grows continuously with an average almost three times as high. It appears that the distribution of exceptions to executions is slightly unbalanced in this case, and should a balanced distribution be a requirement (it is not

```
struct data {
    const static int step = 150;
    const static int scale = 25;
    int length, sum, num;
    vector<int> v;
    void print() {                                          // print nice bar chart
        for ( vector<int> :: iterator it = v.begin(); it != v.end(); it++ ) {
            cout << "|" ;
            for ( int i=0; i < *it / scale; i++ )
                cout << "*";
            std::cout << std::endl;
        }
    }
    void operator++( int ) {
        length++;
        sum += length;
        num++;
        if ( num % step == 0 ) v.push_back( length );       // record every step'th reading
    }
    void operator--( int ) {
        length--;
        sum += length;
        num++;
        if ( num % step == 0 ) v.push_back( length );       // record every step'th reading
    }
};
void averageQueue() {
    using namespace EHMdebug;
    AsyncDebugIter it;
    map< uBaseCoroutine *, data > bch;
    map< unsigned int, uBaseCoroutine * > exh;
    EHMlog_info info;
    for ( ;; ) {
        it >> info;
        switch ( info.type ) {
          case EHMdebug::raise: {
                exh[ info.raise.eID ] = info.raise.exec;    // record exception ID
                data &d = bch[ info.raise.exec ];           // to execution mapping
                d++;                                        // and lengthen execution's queue
                break;
          }
          case EHMdebug::prop: {
                data &d = bch[ exh[ info.prop.eID ] ];      // exception ID -> execution -> data
                d--;                                        // shorten execution's queue
                break;
          }
          case endOfBlock:
             goto end;
        }
    }
  end:
    map< uBaseCoroutine *, data >::iterator finalit = bch.begin();
    for ( ; finalit != bch.end(); finalit++ ) {
        data &d = finalit->second;
        std::cout << "Execution: " << finalit->first
                  <<  ", average queue length: " << ( double ) d.sum / d.num << std::endl;
        d.print();
    }
}
```

Figure 5.3: Calculating exception queues from the asynchronous event log

```
Execution: 0x9c15378, average queue length: 312.273
|******
|************
|******************
|******************
|***************
|*************
|***********
|*******
|****
|
|******
|************
|******************
|************************
Execution: 0x9c1d778, average queue length: 934.501
|*****
|***********
|*****************
|***********************
|*****************************
|*********************************
|*****************************************
|*********************************************
|************************************************
|************************************************************
|*****************************************************************
|*********************************************************************
```

Figure 5.4: Output produced by averageQueue from Figure 5.3

for this particular program), then this result constitutes a first step in recognizing and fixing the problem. A routine like averageQueue can be called from within a debugger like gdb at any point during the execution of the program as many times as required. It can thus serve as a valuable tool while examining the live program during a debugging session.

The routine printAsyncPropControl in Figure 5.5, p. 148 prints out the asynchronous propagation control of all executions at a certain point in the event log specified by a supplied iterator position. First, it iterates over the log until the specified position is reached, and generates an asynchronous-propagation-control stack for each execution using propagation control. Once the final position is reached, it iterates over each such generated stack and calls printAsyncPropHelper, which implements the algorithm from 5.3.2, p. 142 to evaluate the effective propagation-control state. When called at some time in the event log of EHM4.cc with 3 worker tasks[5], printAsyncPropControl produces the following output:

[5]The output shows the propagation control of 5 executions/tasks: 3 worker tasks, plus uMain and uBootTask.

```
Execution 0x80ef190:
-<All>

Execution 0x8b94be8:
-<All>

Execution 0x8b9cfe8:
-rev
+<All>

Execution 0x8ba53e8:
-<All>

Execution 0xbf82aae0:
-<All>
```

This output means that at this specific point in time, execution 0x8b9cfe8 has propagation enabled for all types except rev, whereas all other executions have propagation turned off.

An alternative method to review the log is to dump it in its entirety in human-readable form by using the supplied void EHMdebug::dumpExceptionLog(std::ostream &os, int filter) routine, *e.g.*,

```
EHMdebug::dumpExceptionLog( std::cerr );
```

which produces output similar to:

```
Exception 27 propagated @0x804c853
Execution 0x9feff78: _Disable @[ 0 : 0x804c91c ]
Execution 0x9feff78: _Disable @[ 0 : 0 ]
Execution 0x9feff78: Exception 39 of type rev raised by execution 0x9feff78
Execution 0x9feff78: *last block popped*
Execution 0x9feff78: Exception 39 detected while target task running
Execution 0x9feff78: *last block popped*
...
Execution 0x9ff8378: Exception 48 detected while target task running
Execution 0x9ff8378: *last block popped*
Execution 0x9ff8378: _Enable < uMutexFailure >  @[ 0x8059e2d : 0x8059e2d ]
```

By using the optional filter parameter, only those types of events a programmer is interested in are displayed, *e.g.*,

```
EHMdebug::dumpExceptionLog( std::cout, EHMdebug::prop | EHMdebug::raise );
```

only displays raises and propagations.

Implementation

The major hurdle in implementing the asynchronous event-log is the large amount of data that needs to be stored. This need varies depending on the number of events occurring in a program,

```
using namespace std;
using namespace EHMdebug;
void printAsyncPropHelper( std::ostream &os, vector<DEctor_info > &v ) {
    set< const std::type_info * > array[2];          // array[0] = disabled, array[1] = enabled
    while ( !v.empty() ) {
        DEctor_info &info = v.back();
        int en = info.enabled;
        if ( info.num == 0 ) {
            array[ en ].insert( 0 );                 // once we find the any type, we're done
            break;
        }
      l2: for ( int i = 0; i < info.num; i += 1 ) {
            set<const std::type_info * > :: iterator it = array[1 - en].begin();
            for ( ; it != array[1 - en].end() ; it++ )   // only mark a type as en/disabled if nothing in
                if ( uEHM::match_exception_type( info.pc[i].type, *it ) ) //  the other set matches here
                    continue l2;                     // if there is a match, continue in outer loop
            array[en].insert( info.pc[i].type );     // no match -> insert into the appropriate set
         } // for
        v.pop_back();                                // move on to next entry
    }
    int branch = 1 - array[ 1 ].count( 0 );          // branch 0 -> disabled types first then all enabled
    const char pm[] = { '-','+' };                   // branch 1 -> enabled types first then all disabled
    set<const std::type_info * > :: iterator it = array[ branch ].begin();
    for ( ; it != array[ branch ].end(); it++ ) {
        int status;
        char *s2 = __cxxabiv1::__cxa_demangle( (*it)->name(), 0, 0, &status );
        os << pm[ branch ] << s2 << endl;
        free( s2 );
    }
    os << pm[ 1 - branch ] << "<All>" << endl;
}

void printAsyncPropControl( std::ostream &os, const AsyncDebugIter &it2 ) {
    AsyncDebugIter it;
    map< uBaseCoroutine *, vector< DEctor_info > > m;
    EHMlog_info info;
    for ( ;; ) {
        if ( it == it2 )                             // when point in time denoted by it2
            break;                                   // is reached, leave loop and
        it >> info;
        switch (info.type) {
          case ENctor:
          case DISctor:
            m[ info.ctor.exec ].push_back( info.ctor ); // find execution and push onto stack
            break;
          case dtor:
            m[ info.dtor ].pop_back();               // find execution, pop from stack
            break;
          case endOfBlock:
            return;
        }
    }
    map< uBaseCoroutine *, vector<DEctor_info> >::iterator m_it = m.begin();
    for ( ; m_it != m.end(); m_it++ ) {              // for each execution
        os << "Execution " << m_it->first << ":" << endl;
        printAsyncPropHelper( os, m_it->second );
        os << endl;
    }
}
```

Figure 5.5: Printing executions' propagation control at a given time in the event log

which depends on the frequency of such events during a program's running time. Some test programs that have a large concentration of asynchronous exception-handling activity can produce hundreds of megabyte of raw data per second. The first step to taming this large memory requirement is to recognize that the number of unique events in a program is relatively small compared to the number of times they occur. Executions typically raise a limited number of exception types at a limited number of other executions, and generally, the same code is executed repeatedly. It is therefore useful to aggregate the information that comprises an event, record it in a hash-map, and store the key of the hash-map instead of the raw information. For similar reasons that the same events occur repeatedly, they also occur in certain recurring patterns. This property can be exploited by running the resulting data through a compression algorithm such as zlib's [DG96]. This compression also takes care of space-inefficiencies of data structures employed without having to resort to using unwieldy tools such as bit-fields. In total, the memory requirement can thus be reduced to less than 2% of the raw data in the optimal case (lots of redundancy).

Since the asynchronous event-log is a centralized store, access to it needs to be serialized. This mutual exclusion restricts concurrency in a program and can lead to a performance bottleneck, especially with asynchronous-exception-heavy programs. For example, when collecting full information with the EHM4.cc test program mentioned above, depending on the number of tasks used and whether it is run single-threaded or multi-threaded, the log can cause a slow-down of a factor of 2 up to a factor of 8. The performance can be improved if the type of information collected is restricted using uDefaultEventLogMode(), *e.g.*, with only propagation information, the performance impact for the same test program is undetectable.

5.4 Related Work

A number of publications use static exception-flow analysis to gain static insight into the exceptional control-flow inside a program [SB93, YR97, SH00, RM03]. They analyze a program and determine which exceptions can be raised, what handlers can catch these exceptions, which exceptions can go uncaught, *etc.* These methods give the programmer helpful additional information about the *potential* exceptional control-flow in their programs. Static approaches, however, usually suffer from a tendency to overestimate the number of exceptions that can actually be raised dynamically. These static insights can be used to implement visualization tools that allow the programmer to explore the exception flow in their program through a graphical interface [SGH08]. Dooren and Steegmans [vDS05] try to address the issues of using exception

specifications in practice by introducing the concept of anchored exception handling, which is a way to transitively make a routine's exception specification dependent upon another's.

The concept of assertions can be traced back to [Hoa69], where they are used to describe the pre-/post-conditions of a program execution. Pre-/post-conditions play a major role in Eiffel [Mey87] and its design-by-contract principle. While early versions of Eiffel did not support exception handling, exceptions were added later; in particular, an unmet assertion raises an exception [Mey88]. The extensions of Eiffel to the SCOOP model expand its exception model to allow for exceptions to cross execution boundaries [Ars06]; however, these kinds of exceptions, while the result of asynchronous calls, are only propagated synchronously. Similarly to Eiffel, Berg proposes an exception model where exception raises are the result of unmet assertions [Ber08]. He also proposes a method for dynamically testing exception handling paths in a testing phase of the program, and then using this information to check for coverage of all handlers, as well as to simplify the finding of handlers for given exceptions. Cabral and Marques use a method to inject (synchronous) exceptions, but merely employ it to demonstrate the effectiveness of their automatic exception-handling feature [CM08].

Stack traces are widely used in programming languages, including in Java, Eiffel, Erlang [CGN04], and Python [Lut06]. The C++ standard does not provide a method to generate a stack trace.

POSIX offers no direct method of querying its propagation-control state, but an indirect method can be constructed:

```
int get_pthread_cancelstate() {
    int oldstate;
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);   // set to disable,
                                                                 // and record current state
    pthread_setcancelstate(oldstate, NULL);                      // restore previous state
    return oldstate;                                             // return recorded state
}
```

Note, while it should be benign to temporarily disable cancellation, having to potentially alter the propagation state in order to query it is inelegant.

5.5 Conclusion

This chapter shows exception handling is subject to several usability challenges (see Section 5.1, p. 126). Exception assertions ameliorate these issues in two ways. First, in active mode, exception assertions allow for an increase in propagation frequency and thus better testing. In passive mode,

exception assertions provide a way to codify assumptions about when exceptional propagations occur, which should lead to a better understanding of exception-handling code. The stack-trace feature implemented provides important information about the call stack at the point of propagation, providing a missing element in the C++ language. With regard to the issues specific to asynchronous exceptions, the provided propagation-control and logging features give the programmer additional tools to gain dynamic insight into a program's execution, especially during testing and debugging.

Having these additional capabilities and information is always better then not having them. Programmers can choose to employ them to gain a better understanding of their code. However, while these features' advantages are clear, not all programs may be able to easily exploit them, and not all programmers may actively use them or be able to exploit them to their full potential. As a next step, it would be useful to conduct usability studies of these new features to verify empirically the extent to which they help programmers understand exception handling.

Chapter 6

Conclusion

The following sections provide a summary of insights gained as well as possible future work in the area.

6.1 Summary

In a concurrent system, there is a need for executions to communicate. Communication can be extended to mean asynchronous transfer of control, which is conveniently modeled using the existing exception-handling facilities of a language, leading to the creation of asynchronous exception handling. This asynchronous transfer of control significantly complicates programming as the interruptibility problem is destructive, as well as difficult to recognize and control. This issue mainly affects the full-asynchrony model, but while restricted asynchrony can help in subduing the interruptibility problem, the problem persists. In addition, restricted asynchrony forces an unintuitive programming style using explicit or implicit polls, as well as a polling-related degradation of performance that does not occur with full asynchrony.

Asynchronous propagation control is an essential tool in controlling asynchronous exception handling. Several languages/systems deploy asynchronous propagation control in some form. Without propagation control, safe asynchronous exception handling is virtually impossible, especially with full asynchrony, *e.g.*, as with Java and Thread.stop. There are three basic approaches to propagation control: dynamic, semi-dynamic, and static. The dynamic approach is extremely flexible, whereas the semi-dynamic approach trades in flexibility for increased robustness with regard to programmer error. Both approaches are characterized by an infinite scope; the dynamic approach has an infinite extent whereas the semi-dynamic approach has a finite extent.

Mixing infinite- and finite-extent approaches in the same EHM seems promising at first, but is fraught with difficulties stemming from the need to resolve block- and routine-conflicts. An EHM supporting propagating-control approaches with both kinds of extent that interact with each other seems too complex to use intuitively. Alternatives in which dynamic and semi-dynamic approaches act independently from one another on different exception hierarchies can help, but are complex as well. Further study of this area may be warranted, but for now, using one approach exclusively remains attractive for reasons of simplicity; for μC++, this means a finite-extent approach.

Full asynchrony is difficult to use safely with infinitely-scoped propagation control, which is most commonly used. As a result, restricted asynchrony has to be employed, which is less intuitive along with imposing a delay between the delivery of an asynchronous exception and its propagation. The static approach to propagation control allows for the safe use of full asynchrony. However, static propagation control alone may be too cautious as it does not take advantage of the possibilities available through poll points. A combination approach, where full asynchrony applies within the static scope of an enable block (with regard to the exception types it controls), and restricted asynchrony otherwise, can exploit the advantages of both full asynchrony and restricted asynchrony in a safe way. This combination approach allows for a more intuitive programming style, and can improve performance. However, preexisting code compiled to use the new combination semantics may have to be changed. The provided prototype implementation is a good starting point for implementing such a combination approach as it documents the challenges and pitfalls. It also provides a (limited) vehicle for testing the proposed combination semantics through practical programs. However, it is clear that implementing this approach without compiler-support is difficult and unsuitable for a production system.

A crucial source of delay between delivery and propagation of an asynchronous exception comes from the possibility of a propagating task's being blocked. This possibility can also complicate synchronization protocols. The proposed strategy for transparently unblocking tasks upon an exceptional detection follows directly from the abort characteristics of terminating semantics, and allows the design of unblocking semantics for a multitude of blocking instruments. Pseudo-blocking semantics for resumption unblocking are modelled analogously. The result is a simpler and more intuitive way of writing code, especially with complex synchronization protocols, which can also increase performance. Pseudo-blocking can increase the concurrency in a program, which also improves performance. Exceptional unblocking can prove useful in non-

exceptional areas as well, *e.g.*, to remove tasks from a waiting queue regardless of that queue's priority strategy, or in order to implement message passing. This unblocking functionality has been implemented in μC++, and the theoretical advantages demonstrated in empirical tests. The documentation of the implementation effort can serve as an outline for programmers wishing to adapt unblocking semantics to other languages.

Finally, when adding all these advanced concepts to asynchronous exception handling, it is important to keep usability in mind as well. As shown in the provided references (see Section 5, p. 125), synchronous exception handling is often misunderstood and misused; asynchronous exception handling is even more complex. Attempts to improve usability by statically checked exceptions have had mixed success. Dynamic approaches that allow for better testing may prove more fruitful. The concept of passive exception assertions extends the notion of an assertion into the exception-handling domain. Using these assertions, programmers can codify their assumptions about the exception-handling code of their programs, and are cautioned if these assumptions are not met. In their active variant, exception assertions allow a programmer to test exception-handling code by injecting exceptional propagations at a desired rate, which is especially useful in overcoming the sparse opportunities to test asynchronous exception handling. Giving programmers additional information about what goes on inside a program with regard to (asynchronous) exception handling should lead to better program understanding and better code. Tools like stack traces can give additional information about the call chain during exceptional propagation; insight into the current state of propagation control allows for more flexible adjustment of program behaviour. Finally, the ability to go back and review all important steps affecting asynchronous exception handling should give programmers insight into how their program works, and, most importantly, what causes it to not behave as expected.

6.2 Contributions

A main contribution of this work is identifying the importance of asynchronous propagation control for asynchronous exception handling, as well as analyzing its basic properties and, particularly, its relationship with the asynchrony model employed. While several systems employ propagation control, most do so in an ad-hoc fashion. I am unaware of any work that examines the concept of propagation control at the level of depth and breadth in this thesis.

A further contribution is the novel approach combining full-asynchrony/static-propagation-control with restricted-asynchrony/semi-dynamic-propagation-control. This contribution in-

cludes the theoretical analysis, as well as the demonstration of a limited prototype implementation as a proof-of-concept.

Furthermore, no other work examines the issue of propagating asynchronous exceptions in blocked tasks at the depth and breadth presented herein (other than my previous paper on the same subject). While some systems support limited unblocking semantics, the approach presented provides a theoretical foundation, applies it to a wide variety of blocking instruments, and presents a comprehensive implementation of exceptional unblocking for these instruments, as well as for both termination and resumption semantics.

As a final contribution, this thesis identifies challenges to the usability of (asynchronous) exception handling, proposes solutions to mitigate these challenges, and provides an implementation for these solutions. The concept of exception assertions as presented in this work is unique.

6.3 Future Work

The possibility of combining propagation-control approaches of finite and infinite extent is explored in this work, but no obviously superior strategy is apparent. It might be useful to explore this area further, and implement some of these strategies to gain insight into how they behave with real-world programs.

The prototype implementation for the combination approach to safe full-asynchrony is too limited to be used in a production system. An obvious next step is therefore to augment a compiler with the required functionality in order to overcome these limitations. Improving operating-system components like signal-handler implementations may also be required. An intermediate step is to attack some of the limitations within the prototype implementation itself. New compiler versions may provide new capabilities, *e.g.*, with regard to inlining control.

The theory behind and μC++'s implementation of asynchronous unblocking semantics are at a mature state. Additional testing, *e.g.*, by students in computer science classes, seems a good idea. Porting unblocking semantics to a programming language other than μC++ may yield good results as well since it would underline the generality of the feature and aid in its more wide-spread adoption.

Finally, the implementation of exception assertions is constrained by the current limits of C++. As soon as C++0x is adopted, the described improvements to the implementation can be made. In general, the area of asynchronous-exception usability should be explored further. Asynchronous

exception handling is used so rarely that usability studies are virtually non-existent. Various additional tools are conceivable to assist programmers with asynchronous exception handling. While this thesis concentrates on run-time information, more work should be performed in the compile-time realm. For example, C++'s exception specifications are only checked at run-time, but could be checked at compile-time. This concept could be further extended by integrating the information provided by asynchronous propagation control, and developing an annotation syntax for the asynchronous exception activity of tasks.

Appendix A

Introduction to μC++

μC++ builds on top of C++ to provide object-oriented language-level concurrency, as well as advanced exception-handling features, and other enhancements.

A.1 Control Flow

In μC++, statements can be labelled, and the loop control statements break and continue can be extended by a label to allow for multi-level exit as in this example from [Buh09]:

```
L1: {
    ... declarations ...
    L2: switch ( ... ) {
        L3: for ( ... ) {
            ... break L1; ... // exit compound statement
            ... break L2; ... // exit switch
            ... break L3; ... // exit loop
        }
        ...
    }
    ...
}
```

With nested loops, the following example is possible:

```
outer:
    for (int out = 0; out < 10; out++ ) {
    inner:
        for (int in = 0; in < 10; in++ ) {
            cout << out << endl;
            continue outer;
        }
    cout << "never printed";
    }
```

All that is printed is the outer loop counter from 0 to 9.

A.2 Concurrency

A monitor type is created with type constructor _Monitor, like a Java class where all public members are synchronized, and has all the properties of a C++ class. An active-object type is created with type constructor _Task, like an Ada task type or Java Thread inheritance, and is also a monitor type. An instance of an active-object type must have a main member (analogous to the run member for Java Thread type), where the thread starts execution *implicitly* after the instance is created (no additional call is necessary to start the thread). A task finishes by exiting its main member, and only then can it be destroyed. When a μC++ task, say b, is destroyed, the destroying task, say a joins with it, *i.e.*, a blocks until b finishes and can be destroyed safely. Finally, a μC++ program begins execution in uMain::main rather than main.

```
_Monitor M { ... };             // monitor type, public members are mutex
_Task T {                       // task type
    void main() { ... }         // define action for thread
};
uMain::main() {
    {
        T t;                    // instantiate task and start thread
    }                           // task uMain waits until t is finished
...
}
```

Figure A.1 shows the implementation structure for the mutual exclusion and synchronization of a μC++ monitor. A wait on a condition variable blocks the monitor owner on the specified condition-variable waiting-queue. A signal on a condition variable moves the task at the head of the condition queue (*signallee*) to the signalled stack and the monitor owner (*signaller*) continues. μC++ also supports an _Accept statement to explicitly schedule tasks using rendezvous from outside the monitor. An accept, *e.g.*, _Accept M_1, blocks the owner task on the acceptor stack and the task at the head of the specified mutex (member routine) queue becomes the owner, similar to Ada's select/accept for tasks.[1] In general, the scheduling order is: When the monitor owner exits or waits, the head of the acceptor/signalled-stack (the most recent task signalled or whose accepted member is called) is scheduled (LIFO order). If there is no such task, the head of the entry queue gains ownership (FIFO order, subject to real-time priorities). See [Buh09, §2.9] for more details.

[1] In μC++, the accept concept is generalized across any kind of mutex object, *e.g.*, coroutine, monitor, or task.

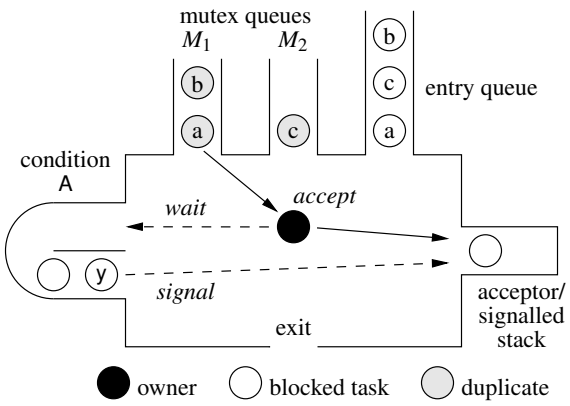

Figure A.1: μC++ monitor

A.3 Exception Handling

An exception type is created with type constructor _Event, which is the only kind of type that can be raised by statements _Throw (termination) and _Resume (resumption). Resumption handlers are defined using a _CatchResume clause[2], which has a mandatory argument referring to the exception object, analogous to the catch clause. Its optional second argument (see example below) specifies the local variables that are referenced within the handler, thus, explicitly capturing a closure. The routine uEHM::poll performs an explicit poll for asynchronous exceptions. Other operations, *e.g.*, entering a monitor, implicitly poll for exceptions (poll points). Asynchronous raise is accomplished by using an _At clause on a raise. Note, asynchronous _Throw has been obsoleted between μC++ versions 5.6.0 and 5.7.0. It is still used in example programs in this document in order to emphasize termination semantics. However, certain parts, *e.g.*, Section 5.2.8, p. 135, do take account of this change. An exception raised with resumption semantics and not handled by a resumption handler has its default resumption handler executed, which by default rethrows the exception using terminating semantics. Hence, an asynchronous _Throw can be achieved by an asynchronous _Resume when there are no user-defined resumption handlers for that exception guarding the detection point.

Asynchronous propagation control is provided using _Enable and _Disable statements. _Enable < T1 > < T2 > means allow propagation of exception types T1 and T2 (as well as their derived types). An _Enable without arguments enables propagation of all exception types.

[2]The _CatchResume statement is new for μC++-5.7.0. Since the test programs in this thesis were compiled using a special version of μC++-5.6.0, they do not actually use this syntax, but rather the older resumption syntax using functors (see Appendix B).

_Disable explicitly disables propagation of exception types with syntax/semantics analogous to _Enable.

```
_Event E {};                        // exception type
...
int a;
_Resume E() _At t;                  // asynchronous throw of exception instance
                                    //     of type E at task t
try {                               // establish handler
    _Enable {                       // enable propagation of any asynchronous exception
        ...uEHM::poll();...         // explicitly poll for asynchronous exceptions
    }
} _CatchResume( E ) ( a ) {}        // resumption handler referring to local variable a
} catch( E ) {}                     // catch termination exception (from _Throw)
```

Unless explicitly enabled, the asynchronous propagation of all exception types are disabled initially. The _Enable statement is a poll point.

Appendix B

Actual Program Code

Several of the programs featured in this thesis use simplified μC++ code for ease of understanding. In order for readers to be able to reproduce the results, the following sections contain the actual programs as compiled and run.

B.1 Time

The Time routine as used in Figure 3.4, p. 84, and Section 3.5, p. 85 returns elapsed computation time per thread in ns, and is defined as

```
#include<sys/time.h>

inline long long Time() {
    timespec ts;

    clock_gettime(CLOCK_THREAD_CPUTIME_ID, &ts);
    return (long long) 1000000000 * ts.tv_sec + ts.tv_nsec;
}
```

B.2 Worry-Free Synchronization

The actual program for Figure 4.6, p. 115 is the following:

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

#define ms * 1000000
_Event CompError {};
_Task Server {
    uBaseTask *c; int run, result, req;
    int compute( int ) throw ( CompError ) {
        _Timeout( uDuration( 0, 50 ms ) ); // work
        if ( ++run % 3 == 0 ) _Throw CompError();
        _Timeout( uDuration( 0, 50 ms ) ); // work
    }
```

```
  public:
    Server() : run( 0 ) {}
    void sendRequest(int n) {c = &uThisTask(); req = n;}
    int getResult() {
#ifdef OLD
        uEHM::poll();
#endif
        return result;
    }
    void main() {
        for ( ;; )
            try <uMutexFailure::RendezvousFailure> {
                _Accept( ~Server ) { break; }
                or _Accept( sendRequest ) {
                    result = compute(req);
                    _Accept(getResult);
                }
            } catch( CompError ) {
                _Throw _At *c;
#ifdef OLD
                try <uMutexFailure::RendezvousFailure> {
                    _Accept( getResult );
                }
#endif
            } // try
    } // main
} server;
_Task Client {
  public:
    void main() {
        for ( int i = 0 ; i < 20 ; i += 1 ) {
            server.sendRequest( i );
            try  {
                _Enable {
                 _Timeout( uDuration( 0, 150 ms) ); // work
                 int res = server.getResult();
                }
            } catch( CompError ) {}
        } // for
    } // main
};
void uMain::main() {
    uProcessor p[2];
    Client c[4];
}
```

B.3 Cheat and Run While Blocked

Figure 4.7, p. 117 is a simplified version of the following program:

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

#define ms * 1000000
const int TASKS=8, CHUNK=10, SPACE=TASKS*CHUNK;
```

```
_Event Recall {
  public:
    int i;  int correction;
    Recall( int i, int c ) : i(i), correction(c) {}
};
_Monitor Mo {
  public:
    void work(int &i) {_Timeout(uDuration(0,100 ms));}
} complete;

void prework(int &d) {_Timeout(uDuration(0,250 ms));}

struct Handler {
    int *data;
    bool *done;
    int &i;

    Handler( int *dat, bool d[], int &i ) : data(dat), done(d), i(i) {}

    void operator() ( Recall &r ) {
        data[r.i] = r.correction;
        prework( data[r.i] );
        complete.work( data[r.i] );
        done[i] = true;
        if ( i == r.i )
            _Throw;
    }
};

_Task Worker {
    int *data;
  public:
    Worker( int * data ) : data( data ) {};
    void main() {
        bool done[CHUNK] = { false };
        int i = 0;
        Handler handler(data, done, i);
        for ( ; i < CHUNK ; i += 1 ) {
            try <Recall, handler> {
                    _Enable {
                        if ( done[i] ) continue; // iteration fixed ?
                        prework( data[i] );
                        uEHM::poll();
                        complete.work( data[i] );
                    }

                } catch ( Recall ) { }
        } // for
    } // main
};

void uMain::main() {
    uProcessor p[TASKS];                    // create kernel thread per worker
    int space[TASKS][CHUNK];                // space for each worker
    Worker *w[TASKS];                       // workers
    for ( int i = 0; i < TASKS; i += 1 )
        w[i] = new Worker( space[i] );      // create worker tasks and start running
    for ( int i = 0; i < TASKS * CHUNK; i += 4 ) { // fix every 4th chunk across all tasks
        _Timeout( uDuration( 0, 100 ms ) ); // delay before next recall
```

```
            _Resume Recall( i % CHUNK, 3 ) _At *w[i / CHUNK];
        }
        for ( int i = 0; i < TASKS; i += 1 )
            delete w[i];                                    // wait for worker tasks to finish
} // main
```

B.4 Sleeping Barber

Here is the complete sleeping barber program from Figure 4.9, p. 121:

```
#include <uC++.h>
#include <iostream>
using namespace std;

_Task Customer;
Customer *customers[20];                                    // customers indexed by id

class BarberShop {
    _Event Wake {};

    _Monitor WaitingRoom {
        const int max;
        int count, next;
        int *chairs;
      public:
        WaitingRoom( int m ) : max(m), count(0), next(0) {
            chairs = new int[max];
        }
        ~WaitingRoom() { delete [] chairs; }
        bool isSpace( int id ) {
            cout << "isSpace " << id << " " << count << " " << max << endl;
          if ( count == max ) return false;
            chairs[ (next + count) % max ] = id;
            count += 1;
            return true;
        }
        int getNext();
    } w;

    _Monitor WaitingChairs {
      public:
        void dummy() {}        // never called
        void waitTurn() { _Accept( dummy ); }
    } waitingChairs;

    _Monitor BarberChair {
        bool ready;
      public:
        BarberChair() : ready( false ) {}
        void sync() {
            ready = ! ready;
            if ( ready ) _Accept( sync );
        }
    } barberChair;
  public:
    BarberShop( int max ) : w( max ) {}
    bool hairCut( int id ) {                                // called by customer
```

```
        if ( ! w.isSpace( id ) ) return false;                  // balk ?
        if ( id == -1 ) return true;                            // program shutting down
        cout << "hairCut " << id << endl;
        try {
            _Enable <Wake> { waitingChairs.waitTurn(); }
        } catch ( Wake ) {  barberChair.sync(); barberChair.sync(); }
        return true;
    }
    int startCut() {                                            // called by barber
        int id = w.getNext();
        if ( id == -1 )
            return id;
        barberChair.sync( );
        return id;
    }
    void endCut() { barberChair.sync( ); }                      // called by barber
} shop( 10 );

_Task Customer {
    unsigned int id;                                            // task identifier

    void main() {
        osacquire( cout ) << id << " " << shop.hairCut( id ) << endl; // get hair cut ?
    } // Customer::main
  public:
    Customer( unsigned int id ) : id( id ) {}
}; // Customer

int BarberShop::WaitingRoom::getNext() {
            if ( count == 0 ) _Accept( isSpace );
            int id = chairs[next];                              // id of next customer
            count -= 1;
            next = (next + 1) % max;
            if ( id != - 1 ) _Throw Wake() _At *customers[id];   // unblock customer
            return id;
        }

_Task Barber {
    void main() {
        for ( ;; ) {
            int custId = shop.startCut();                       // get customer
            yield ( rand() % 5 );
            if ( custId == -1 ) break;
          shop.endCut();                                        // release customer
        } // for
    } // Barber::main
}; // Barber

void uMain::main() {
    Barber barber;

    for ( unsigned int i = 0; i < 20; i += 1 ) {
        customers[i] = new Customer( i );
    } // for
    for ( unsigned int i = 0; i < 20; i += 1 ) {
        delete customers[i];
    } // for
    shop.hairCut( -1 );                                         // tell barber to ghome
}
```

B.5 Exception Assertion Example

The following program is the complete version for that found in Figure 5.1, p. 134:

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

const int N = 20;

_Event GotInput {
  public:
    int input;
    GotInput( int i ) : input( i ) {}
};

_Event FinishedCalc {
  public:
    void *result;
};
_Event End {};

void outputResult( void * ) {}

class Server {
  public:
    void process( int input ) {}
} server[ N ];

class GIh {
    int &outstanding;
  public:
    GIh( int &i ) : outstanding(i) {}
    void operator () ( GotInput &gi ) {
        if ( gi.input == 0 )
            _Throw End();
        outstanding++;
        server[rand() % N].process( gi.input );
    }
};

class FCh {
    int &outstanding;
  public:
    FCh( int &i ) : outstanding(i) {}
    void operator () ( FinishedCalc &fc ) {
        outstanding--;
        outputResult( fc.result );
    }
};

void eventLoop() {
    int outstanding = 0;
    GIh gih(outstanding);
    FCh fch(outstanding);

    try < GotInput, gih > < FinishedCalc, fch > {
            for (; ;) {
                _Enable < GotInput( rand() %  (N) )  ? (outstanding < N), .75 >
                    < FinishedCalc ? (outstanding > 0), 0.25 >  {
```

```
                        std::cout << outstanding << "  ";
                    }
                }
            } catch ( End) {
            std::cout << std::endl;
        }
}
void uMain::main() {
    eventLoop();
}
```

Appendix C

Exception Assertion Transformation

The following is an example for the transformation performed in order to facilitate passive and active assertion checking.

C.1 Original Code

In the routine test in the following code, variable c basically counts how many exceptions are injected in total:

```
_Event mary {
    int i;
  public:
    mary ( int i ) : i ( i ) {}
};
double getAssertProbability( double probability, int group ) {
    return probability / group;
}
void foo() {}

void test() {
    int a = 0, b = 1, c = 0, i;

    for ( i = 0; i < 1000; i++, b *= -1) {
        try < mary( b ) ? ( b > a ), 0.6, 2 > {
                a += 0;                                  // no-effect statement
                foo();                                   // no-effect statement
                a -= 0;                                  // no-effect statement
        } catch( mary ) { c++; }
    }
}
```

The correct final value for c should cluster around 150 as the probability for assertions of group 2 is halved (from 0.6 to 0.3 in this case) and the assertion condition only holds for half the loop iterations.

C.2 Transformation

When both active and passive exception assertion checking is enabled, routine test from the code above is transformed into the following code (comments added for clarity). Note that uEHM::triggerAssertion calls getAssertProbability to get the effective probability for an assertion, calculates the individual probability for each sub-trigger, and performs a random draw to determine whether the active assertion is triggered, *i.e.*, an exception raised.

```
struct _fnc0xb707e8f4 {                                    // the handler functor
    const bool cond;                                       // stores the value of the condition on block entry
    void operator () ( mary &ex ) {                        // handler routine
    if ( !cond )
        std::osacquire( std::cerr ) << "Warning: On _Enable block entry: Assertion ( b > a ) false
\for raised "
                                    << ( &ex ? "resumption" : "exception" ) << " of type mary "
                                    << std::endl;
    if (&ex) uEHM::ReResume();                             // ex != NULL means its a resumption => reraise
    }
    _fnc0xb707e8f4 ( bool b ) : cond ( b ) {}
};

void test ( ) {

    int a = 0 , b = 1 , c = 0 , i ;

    for ( i = 0 ; i < 1000 ; i ++ , b *= - 1 ) {
        {
            try {
        // instantiate handler functor, evaluate condition (for passive assertion)
                _fnc0xb707e8f4 _inst0xb707e8f4 ( b > a ) ;
        // set up exception -> handler mapping
                uHandlerClause < mary , typeof ( _inst0xb707e8f4 ) > _uH_inst0xb707e8f4 ( ( void * ) 0,
                    _inst0xb707e8f4 ) ;
                uEHM :: uHandlerBase * _uT_inst0xb707e8f4 [ ] = { & _uH_inst0xb707e8f4 ,} ;
                uEHM :: uResumptionHandlers _uRN_inst0xb707e8f4 ( _uT_inst0xb707e8f4 , 1);
        // remember total number of sub-triggers
                const int _uTotalNoTriggers = 3 ;
                try { {
                    a += 0 ;
        // inserted sub-trigger (active assertion)
                    {
        // remember number of this sub-trigger
                        const int _uTheAssertionSubTrigger = 0 ;
        // if condition met and probabilistic triggerAssertion fires,
        // resume the exception with the instantiation as specified
                        if ( ( b > a ) && uEHM::triggerAssertion( _uTotalNoTriggers,
                                _uTheAssertionSubTrigger , 0.6 , 2 ) )
                            mary ( b ) . setOriginalThrower( this ).Resume();
                    }
                    foo ( ) ;
        // inserted sub-trigger (active assertion)
                    {
        // remember number of this sub-trigger
```

```
                    const int  _uTheAssertionSubTrigger = 1 ;
        // if condition met and probabilistic triggerAssertion fires,
        // resume the exception with the instantiation as specified
                    if ( ( b > a ) &&  uEHM::triggerAssertion( _uTotalNoTriggers,
                            _uTheAssertionSubTrigger , 0.6 , 2 ) )
                        mary ( b ) . setOriginalThrower( this ).Resume();
                }
                a -= 0 ;
        // inserted sub-trigger (active assertion)
                {
        // remember number of this sub-trigger
                    const int  _uTheAssertionSubTrigger = 2 ;
        // if condition met and probabilistic triggerAssertion fires,
        // resume the exception with the instantiation as specified
                    if ( ( b > a ) &&  uEHM::triggerAssertion( _uTotalNoTriggers,
                            _uTheAssertionSubTrigger , 0.6 , 2 ) )
                        mary ( b ) . setOriginalThrower( this ).Resume();
                }
            } } catch ( mary ) {                        // termination handler (for passive assertion)
                _inst0xb707e8f4 ( *( mary  *) 0 );      // invoke handler inside functor
                if (! ( b > a ) ) {                     // check condition (on propagation out of block)
                    std::osacquire( std::cerr ) << "Warning: Assertion  ( b > a ) failed for \
    exception raise of type  mary " << std::endl;
                }
                throw;                                  // reraise exception
            }
        } catch ( mary ) {                              // pre-existing handler
            { c ++ ; }
        }
    }
  }
}
```

Appendix D

Asynchronous Event Log Specifications

All classes and routines required to access the asynchronous event log from Section 5.3.3, p. 143 are declared within the EHMdebug.h header file and reside within the EHMdebug name-space[1]. Class EHMdebug::AsyncDebugIter provides a forward iterator to traverse the log. When an AsyncDebugIter object is instantiated (its constructor needs no arguments) it points before the first entry of the log. Two AsyncDebugIter iterators can be compared using the == operator.

Retrieving log information follows a two-step process: First, identifying the type of log entry stored at the next iterator location, and second, retrieving the information from the log entry into a structure of suitable type.

D.1 Data Types and Operations

The enum EHMdebug::type_e defines the following types of information:

```
enum type_e { raise, detect, prop, ENctor, DISctor, dtor, endOfBlock };
```

which signify raise, detection, propagation, _Enable block establishment, _Disable block establishment, _Enable/_Disable block destruction, and the end of the log, respectively. The iterator is advanced to the next log entry and its type queried using the >> (type_e &) operator, *e.g.*:

```
EHMdebug::AsyncDebugIter it;
EHMdebug::type_e t;
it >> t;                          // advance to next entry and store type of log entry in t
```

Note, that using the >> (type_e &) operator only advances the iterator and fills in the type and does not transfer actual information from the log. The data structures that accept the appropriate

[1] Assume for the following examples that they are preceded by a using namespace EHMdebug.

information from the log are declared as follows:

```
struct Raise_info {                     /* Event: Asynchronous raise */
    const std::type_info *type;             /* Type of raised exception */
    uBaseCoroutine *source;                 /* Address of raising execution */
    uBaseCoroutine *exec;                   /* Address of propagating execution */
    unsigned int eID;                       /* Unique ID of exception */
};

struct Detect_info {                    /* Event: Detection, consider exception for propagation */
    unsigned int eID;                       /* Unique ID of exception */
    uBaseCoroutine *exec;                   /* Address of detecting execution */
    uBaseTask::State targetState;           /* State of thread of the propagating execution */
};

struct Prop_info {                      /* Event: Propagation */
    unsigned int eID;                       /* Unique ID of exception */
    void * address ;                        /* Instruction address at which propagation occurs */
};

struct DEctor_info {                    /* Event: Additional layer of propagation control pushed */
    bool enabled;                           /* true => _Enable, false => _Disable */
    uBaseCoroutine *exec;                   /* Execution to which propagation control applies */
    void *start;                            /* Start of _Enable/_Disable block (if available) */
    void* end;                              /* End of _Enable/_Disable block (if available) */
    int num;                                /* Number of types controlled */
    const uEHM::PropControl* pc;            /* Pointer to a PropControl[num] array of types */
};
                                        /* Event: Top layer of propagation control popped */
typedef uBaseCoroutine * DEdtor_info;   /* Execution to which propagation control applies */
```

where the association between type and corresponding data structure is obvious. Note that DEctor_info applies to both ENctor and DIScto r types. DEctor_info makes use of the structure PropControl, which is declared as (some parts omitted):

```
struct PropControl {
    uBaseCoroutine *binding;
    const std::type_info *type;
    ...
};
```

where binding takes the address of a bound-to object while type points to the type_info of the exception type controlled. For example, for a _Disable < client1.PacketLoss >< Shutdown > block, the corresponding DEctor_info structure has (apart from enabled being false) a value of 2 for num and pc pointing to a two-member array, where the following holds:

```
pc[0].binding == &client1               &&
pc[0].type == &typeof(PacketLoss)       &&
pc[1].binding == NULL                   &&
pc[1].type == &typeof(Shutdown)
```

Note that std::type_info is declared in the standard C++ header file typeinfo.

For convenience, the types above are organized as

```
struct EHMlog_info {
    type_e type;
    union {
        Raise_info raise;
        Detect_info detect;
        Prop_info prop;
        DEctor_info ctor;
        DEdtor_info dtor;
    };
};
```

and operator >> (EHMlog_info &) is provided to advance the iterator to the next position and fill the appropriate data structure with the information from the log. Note that when traversing the log without retrieving information (*e.g.*, in order to skip uninteresting log entries), this operator is less efficient than the >> (type_e &) operator, *e.g.*,

```
do {
    it >> t;                    // t is still of type type_e
} while ( t != prop );          // advance through log until a propagation is found
```

but in most cases, performance of log traversal should be irrelevant.

D.2 Information Retrieval

Typically, the log information is processed in a loop where first the iterator is advanced to the next entry as well as the appropriate data structure filled with the log information, and then a switch statement chooses the right structure to extract the information, *e.g.*,

```
EHMdebug::AsyncDebugIter it;
EHMdebug::EHMlog_info info;
for (;;) {
    it >> info;                 // advance iterator and fill in log information
    switch ( info.type ) {
      case ENctor:
      case DISctor:
          /* do something with the information stored in info.ctor */
          break;
      case detect:
          /* do something with the information stored in info.detect */
          break;
      ...
      case endOfBlock:
        return;                 // end of log information reached, done
    }
}
```

For more elaborate examples of traversing the event log and processing its information, see Section 5.3.3, p. 144.

D.3 Information Collection

Collection of log information is controlled by uDefaultEventLogMode(), whose default implementation is

```
int uDefaultEventLogMode() {
    return 0;
}
```

and collects no information. By redefining this routine and returning the desired combination of type_e types, additional types of information can be collected, *e.g.*,

```
int uDefaultEventLogMode() {
    return EHMdebug::prop | EHMdebug::detect;
}
```

only collects information about propagations and detections. Returning -1 is a simple way to enable the collection of all information.

Bibliography

[Abr98] David Abrahams. Exception-safety in generic components. In *Generic Programming*, pages 69–79, 1998. 9, 97

[AOC+88] Gregory R. Andrews, Ronald A. Olsson, Michael Coffin, Irving Elshoff, Kelvin Nilsen, Titus Purdin, and Gregg Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988. 119

[Ars06] Volkan Arslan. Asynchronous exceptions in concurrent objectoriented programming. In Richard F. Paige and Phillip J. Brooke, editors, *Symposium on Concurrency, Real-Time, and Distribution in Eiffel-Like Languages*, pages 62–70, 2006. 150

[BBD+00] Greg Bollella, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, and Mark Turnbull. *The Real-Time Specification for Java*. The Real-Time for Java Expert Group, http://www.rtj.org. Addison-Wesley, 2000. 24

[Ber08] Bradley A. Berg. Disentangling exceptions. Master's thesis, Brown University, 2008. 150

[BFC95] Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, March 1995. 98

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14:317–329, November 2000. 71

[BHLC00] Peter A. Buhr, Ashif S. Harji, Philipp E. Lim, and Jiongxiong Chen. Object-oriented real-time concurrency. *SIGPLAN Notices*, 35(10):29–46, October 2000. OOPSL'00, Oct. 15–19, 2000, Minneapolis, Minnesota, U.S.A. 103

[BHM02] Peter A. Buhr, Ashif Harji, and W. Y. Russell Mok. Exception handling. In Marvin V. Zelkowitz, editor, *Advances in COMPUTERS*, volume 56, pages 245–303. Academic Press, 2002. 9, 10, 17, 127

[BHR02] Benjamin M. Brosgol, Ricardo J. Hassan, II, and Scott Robbins. Asynchronous transfer of control in the real-time specification for Java™. In *IRTAW '02: Proceedings of the 11th international workshop on Real-time Ada workshop*, pages 95–112, New York, NY, USA, 2002. ACM. 16, 66

[Bir89] Andrew D. Birrell. An introduction to programming with threads. Technical Report 35, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California, 94301, January 1989. 15, 93

[BK06] Peter A. Buhr and Roy Krischer. Bound exceptions in object-oriented programming. In *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2006. 108, 122

[BM00] Peter A. Buhr and W. Y. Russell Mok. Advanced exception handling mechanisms. *IEEE Transactions on Software Engineering*, 26(9):820–836, September 2000. 16, 17, 18, 19, 107

[BMZ92] Peter A. Buhr, Hamish I. Macdonald, and C. Robert Zarnke. Synchronous and asynchronous handling of abnormal events in the μSystem. *Software—Practice and Experience*, 22(9):735–776, September 1992. 19

[BR86] Theodore P. Baker and Gregory A. Riccardi. Implementing ada exceptions. *IEEE Software*, 3(2):42–51, 1986. 77

[Bri73] Per Brinch Hansen. Concurrent programming concepts. *Software—Practice and Experience*, 5(4):223–245, December 1973. 27

[Buh09] Peter A. Buhr. μC++ annotated reference manual, version 5.6.0. Technical report, January 2009. http://plg.uwaterloo.ca/~usystem/pub/uSystem/u++-5.6.0.pdf. 80, 159, 160

[But97] David R. Butenhof. *Programming with POSIX Threads*. Professional Computing. Addison-Wesley, 1997. 16, 21, 49

[BW97] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, second edition, 1997. 6, 15

[BW03] Benjamin M. Brosgol and Andy Wellings. A comparison of the asynchronous transfer of control features in ada and the real-time specification for java. In *Ada and the Real-Time Specification for Java, Proc. Ada Europe 2003*, 2003. 15

[CC05] Denis Caromel and Guillaume Chazarain. Robust exception handling in an asynchronous environment. In *ECOOP Workshop on Exception Handling in Object-Oriented Systems: Developing Systems that Handle Exceptions, number 05050 in Technical Reports - Laboratoire*, 2005. 14

[CG92] Qian Cui and John Gannon. Data-oriented exception handling. *IEEE Transactions on Software Engineering*, 18(5):393–401, May 1992. 126

[CGN04] Richard Carlsson, Björn Gustavsson, and Patrik Nyblom. Erlang's exception handling revisited. In *Proceedings of the 2004 ACM SIGPLAN workshop on Erlang*, ERLANG '04, pages 16–26, New York, NY, USA, 2004. ACM. 150

[Cha94] David Chase. Implementation of exception handling, part II, calling conventions, asynchrony. optimizers, and debuggers. *Journal of C Language Translation*, oct 1994. 12

[CM08] B. Cabral and P. Marques. A case for automatic exception handling. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 403–406, Washington, DC, USA, 2008. IEEE Computer Society. 129, 150

[CR86] Roy H. Campbell and Brian Randell. Error recovery in asynchronous systems. *IEEE Trans. Softw. Eng.*, 12(8):811–826, 1986. 14, 18, 49

[Cri82] F. Cristian. Exception handling and software fault tolerance. *Computers, IEEE Transactions on*, C-31(6):531 –540, jun. 1982. 17

[DG96] P. Deutsch and J-L. Gailly. ZLIB Compressed Data Format Specification version 3.3. RFC 1950 (Informational), May 1996. 149

[DGL95] S. Drew, K. Gouph, and J. Ledermann. Implementing zero overhead exception handling. Technical report, Faculty of Information Technology, Queensland University of Technology, 1995. TR 95-12. 77

[Dij65] Edsger W. Dijkstra. Cooperating sequential processes. Technical report, Technological University, Eindhoven, Netherlands, 1965. Reprinted in [Gen68] pp. 43–112. 97, 118

[Don01] Chistophe Dony. A fully object-oriented exception handling system: Rationale and smalltalk implementation. In *Exception Handling*, volume 2022 of *Lecture Notes in Computer Science*, pages 18–38. Springer-Verlag, 2001. 107

[Duf07] Joe Duffy. Monitor.enter, thread aborts, and orphaned locks. *Joe Duffy's Weblog*, 2007. WWW, Nov 8 2010, http://www.bluebytesoftware.com/blog/2007/01/30/MonitorEnterThreadAbortsAndOrphanedLocks.aspx. 20

[DUV06] Christophe Dony, Christelle Urtado, and Sylvain Vauttier. Exception handling and asynchronous active objects: Issues and proposal. In C. Dony, J. L. Knudsen, A. Romanovsky, and A. Tripathi, editors, *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 81–100. Springer-Verlag, 2006. 10.1007/11818502_5. 18

[Eck07] Bruce Eckel. Does Java need checked exceptions?, 2007. WWW, Sep 13 2010, http://www.mindview.net/Etc/Discussions/CheckedExceptions. 9, 10, 125, 127

[Eri] Ericsson AB. Processes. In *Erlang Reference Manual User's Guide Version 5.8.1*. WWW, Nov 6 2010, http://www.erlang.org/doc/reference_manual/processes.html. 15

[Fee93] Marc Feeley. Polling efficiently on stock hardware. In *in Proceedings of the 1993 ACM SIGPLAN Conference on Functional Programming and Computer Architecture*, pages 179–187, 1993. 53, 57, 92

[FFM+10] Gerhard Friedrich, Mariagrazia Fugini, Enrico Mussi, Barbara Pernici, and Gaston Tagni. Exception handling for repair in service-based processes. *IEEE Transactions on Software Engineering*, 99(RapidPosts):198–215, 2010. 18

[FFS96] Claudio Fleiner, Jerry Feldman, and David Stoutamire. Killing threads considered dangerous. In *POOMA 96: Conference on Parallel and Object Oriented Methods and Applications*, 1996. WWW, Dec 21 2010, http://www.icsi.berkeley.edu/∼sather/-Publications/pooma96-text.html. 14, 15, 93

[Geh92] N. H. Gehani. Exceptional C or C with exceptions. *Software—Practice and Experience*, 22(10):827–848, October 1992. 17, 107

[Gen68] F. Genuys, editor. *Programming Languages*. Academic Press, New York, 1968. NATO Advanced Study Institute, Villard-de-Lans, 1966. 182

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000. 9, 127, 140

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005. 20, 94

[Goo75] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975. 9, 17, 107

[GR89] N. H. Gehani and W. D. Roome. *The Concurrent C Programming Language*. Silicon Press, NJ, 1989. 119

[Hei03] Anders Heijlsberg. The trouble with checked exceptions. In Bill Venners and Bruce Eckel, editors, *A Conversation with Anders Hejlsberg, Part II*, August 2003. WWW, Sep 13 2010, http://www.artima.com/intv/handcuffs.html. 127

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. 150

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974. 98, 99, 101

[How76] J. H. Howard. Signaling in monitors. In *Proceedings Second International Conference Software Engineering*, pages 47–52, San Francisco, U.S.A, October 1976. IEEE Computer Society. 98

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. 110

[IEE01] IEEE and The Open Group. *1003.1 Standard for Information Technology – Portable Operating System Interface (POSIX), System Interface, Issue 6*, 2001. 5, 12, 16, 21

[Int95] Intermetrics, Inc. *Annotated Ada Reference Manual*, international standard ISO/IEC 8652:1995(E) with COR.1:2000 edition, December 1995. Language and Standards Libraries. 15, 93

[Int98] International Standard ISO/IEC 14882:1998 (E), www.ansi.org. *Programming Languages – C++*, 1998. 80, 139

[ISO10] ISO/IEC IS 14882:2010 (Working Draft N3092), ISO copyright office, Case postale 56, CH-1211 Geneva 20. *Programming Languages – C++*, 2010. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf. 137

[Iss91] Valérie Issarny. An exception handling model for parallel programming and its verification. In *SIGSOFT '91: Proceedings of the conference on Software for citical systems*, pages 92–100, New York, NY, USA, 1991. ACM Press. 14, 49

[IY91] Yuuji Ichisugi and Akinori Yonezawa. Exception handling and real time features in an object-oriented concurrent language. In *Proceedings of the UK/Japan workshop on Concurrency : theory, language, and architecture*, pages 92–109, New York, NY, USA, 1991. Springer-Verlag New York, Inc. 14

[KB08] Roy Krischer and Peter A. Buhr. Asynchronous exception propagation in blocked tasks. In *4th International Workshop on Exception Handling (WEH.08)*, pages 8–15, Atlanta, U.S.A, November 2008. 16th International Symposium on the Foundations of Software Engineering (FSE 16). 91

[Knu84] Jørgen Lindskov Knudsen. Exception handling — a static approach. *Software—-Practice and Experience*, 14(5):429–449, May 1984. 17

[Knu87] Jørgen Lindskov Knudsen. Better exception handling in block structured systems. *IEEE Software*, 4(3):40–49, May 1987. 128

[KO02] Aaron Keen and Ronald Olsson. Exception handling during asynchronous method invocation. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par 2002 Parallel Processing*, volume 2400 of *Lecture Notes in Computer Science*, pages 337–412. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45706-2_90. 14

[Kri02] Roy Krischer. Bound exceptions in object-oriented programming languages. Diplomarbeit, Universität Mannheim, Mannheim, Deutschland, October 2002. ftp://-plg.uwaterloo.ca/pub/theses/KrischerThesis.ps.gz. 17

[Kri09a] Roy Krischer. Re: libunwind bug. In *Developer's mailing list for the libunwind library*. WWW, Oct 20 2010, http://permalink.gmane.org/gmane.comp.lib.unwind.devel/431, 2009. 81

[Kri09b] Roy Krischer. uw_frame_state_for can segfault when called by _unwind_backtrace from asynchronous signal handler. In *GCC Bugzilla*. WWW, Oct 20 2010, http://gcc.gnu.org/bugzilla/show_bug.cgi?id=40466, 2009. 80

[KS90] Andrew Koenig and Bjarne Stroustrup. Exception handling for C++. *Journal of Object-Oriented Programming*, 3(2):16–33, July/August 1990. 17

[KS93] Andrew Koenig and Bjarne Stroustrup. *Exception handling for C++*, pages 137–171. MIT Press, Cambridge, MA, USA, 1993. 77

[KT10] Devdatta Kulkarni and Anand Tripathi. A framework for programming robust context-aware applications. *IEEE Transactions on Software Engineering*, 99(RapidPosts):184–197, 2010. 18

[Lib] Libunwind library. WWW, Oct 20 2010, http://www.nongnu.org/libunwind/. 81

[LS79] Barbara H. Liskov and Alan Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, SE-5(6):546–558, November 1979. 9, 17, 107

[LS98] Jun Lang and David B. Stewart. A study of the applicability of existing exception-handling techniques to component-based real-time software technology. *ACM Transactions on Programming Languages and Systems*, 20(2):274–301, March 1998. 17

[Lut06] Mark Lutz. *Programming Python*. O'Reilly Media, Inc., 2006. 150

[Mar06] Simon Marlow. An extensible dynamically-typed hierarchy of exceptions. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, Haskell '06, pages 96–106, New York, NY, USA, 2006. ACM. 21

[Mey87] B Meyer. Eiffel: programming for reusability and extendibility. *SIGPLAN Not.*, 22:85–94, February 1987. 150

[Mey88] Bertrand Meyer. Disciplined exceptions. Technical Report TR-EI-13/EX, Interactive Software Engineering, 1988. 150

[Mica] Microsoft. *Microsoft Visual Studio 2008/.NET Framework 3.5 Documentation: ThreadAbortException Class*. WWW, Oct 20 2010, http://msdn.microsoft.com/en-us/library/system.threading.threadabortexception.aspx. 16

[Micb] Microsoft. .NET Framework 1.1, StackTrace class. In *Microsoft Development Network Library*. 140

[MJMR01] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in haskell. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 274–285, New York, NY, USA, 2001. ACM. 3, 12, 15, 20, 21, 34, 75, 94

[Mok97] Wing Yeung Russell Mok. Concurrent abnormal event handling mechanisms. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, September 1997. ftp://plg.uwaterloo.ca/pub/theses/MokThesis.ps.gz. 137

[MT02] Robert Miller and Anand Tripathi. The guardian model for exception handling in distributed systems. *Reliable Distributed Systems, IEEE Symposium on*, 0:304, 2002. 14, 18, 21, 49

[PRP04] Christian Pérez, André Ribes, and Thierry Priol. Handling exceptions between parallel objects. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing*, volume 3149 of *Lecture Notes in Computer Science*, pages 671–678. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-27866-5_88. 14

[Rin06] Matti Rintala. Handling multiple concurrent exceptions in C++ using futures. In *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 62–80. Springer-Verlag, 2006. 14, 49

[RM03] Martin P. Robillard and Gail C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.*, 12(2):191–221, 2003. 149

[RS03] Barbara G. Ryder and Mary Lou Soffa. Influences on the design of exception handling acm sigsoft project on the impact of software engineering research on programming language design. *SIGSOFT Softw. Eng. Notes*, 28(4):29–35, 2003. 17

[RXR98a] A. Romanovsky, J. Xu, and B. Randell. Exception handling and resolution in distributed object oriented systems. pages 545–553, 1998. 18

[RXR98b] A. Romanovsky, J. Xu, and B. Randell. Exception handling in object-oriented real-time distributed systems. In *ISORC '98: Proceedings of the The 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 32, Washington, DC, USA, 1998. IEEE Computer Society. 14, 49

[SB93] Carl F. Schaefer and Gary N. Bundy. Static analysis of exception handling in ada. *Software - Practice and Experience*, 23(10):1157–1174, 1993. 149

[SGH08] Hina Shah, Carsten Görg, and Mary Jean Harrold. Visualization of exception handling constructs to support program understanding. In *Proceedings of the 4th ACM symposium on Software visualization*, SoftVis '08, pages 19–28, New York, NY, USA, 2008. ACM. 149

[SGH10] Hina B. Shah, Carsten Görg, and Mary Jean Harrold. Understanding exception handling: Viewpoints of novices and experts. *IEEE Transactions on Software Engineering*, 99(RapidPosts):150–161, 2010. 13, 125, 126

[SH00] Saurabh Sinha and Mary Jean Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Trans. Softw. Eng.*, 26:849–871, September 2000. 149

[SS85] Andrzej Szalas and Danuta Szczepanska. Exception handling in parallel computations. *SIGPLAN Notices*, 20(10):95–104, October 1985. 14, 22

[Sta] Richard M. Stallman. *GCC*. Free Software Foundation, Cambridge, MA. 65

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994. 17, 98, 107

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997. 9, 17, 45

[Sun] Sun Microsystems. java.lang.Thread. In *Java™ 2 Platform Standard Ed. 5.0 Documentation*. WWW, Nov 9 2010, http://download.oracle.com/javase/1.5.0/docs/api/-java/lang/Thread.html. 16, 20

[TCMM00] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test-data generation for exception conditions. *Softw. Pract. Exper.*, 30(1):61–79, 2000. 129

[vDS05] Marko van Dooren and Eric Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. *SIGPLAN Not.*, 40(10):455–471, 2005. 149

[Wal72] David C. Walden. A system for interprocess communication in a resource sharing computer network. *Commun. ACM*, 15(4):221–230, 1972. 122

[Wei06] Westley R. Weimer. Exception handling bugs in Java and a language extension to avoid them. In C. Dony, J. L. Knudsen, A. Romanovsky, and A. Tripathi, editors, *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 22–41. Springer-Verlag, 2006. 10

[YB85] Shaula Yemini and Daniel M. Berry. A modular verifiable exception-handling mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2):214–243, April 1985. 17

[YR97] Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Static Analysis Symposium*, pages 98–113, 1997. 149

[ZHR+] Sharon Zakhour, Scott Hommel, Jacob Royal, Isaac Rabinovitch, Tom Risser, and Mark Hoeber. *The Java™ Tutorials*. Sun Microsystems. WWW, Sep 13 2010, http://java.sun.com/docs/books/tutorial/. 10