Bound Exceptions in Object-Oriented Programming Languages

Diplomarbeit

Von

Roy Krischer

aus

Bensheim

vorgelegt am

Lehrstuhl für Praktische Informatik IV

Prof. Dr. W. Effelsberg

Fakultät für Mathematik und Informatik

Universität Mannheim

Oktober 2002

Betreuer: Prof. Dr. Wolfgang Effelsberg

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Waterloo, den 14. Oktober 2002 Roy Krischer

Abstract

Modern object-oriented programming languages provide Exception Handling Mechanisms (EHMs) to deal with exceptional situations during program execution. However, exception handling is still detached from the object-oriented design of a program since current EHMs cannot associate exceptions with objects. There have been attempts to associate exceptions with data structures (through data-oriented exception handling), but those solutions lack true object-oriented functionality.

This thesis examines the concept of truly object-oriented Bound Exceptions. The motivation for their use is explained, as well as methods of mimicking their behaviour using conventional EHMs. Different binding strategies are presented, including Dynamic Binding, which strengthens the object-oriented design with Bound Exceptions. A Bound Exceptions design for the programming language C++ is presented and extended to support the concurrency and resumption features of μC++. Following this design, an implementation of Bound Exceptions as part of μC++'s EHM is described.

Acknowledgements

First and foremost, I would like to thank my supervisor at the University of Waterloo, Professor Dr. Peter Buhr, for giving me the opportunity to write this thesis. It was his guidance and support that made this work possible, and for this, I am very grateful.

Furthermore, I would like to thank my supervisor at the University of Mannheim, Professor Dr. Wolfgang Effelsberg, who allowed me to come to Waterloo in the first place and whose willingness to supervise my thesis allowed me to stay here.
Additional thanks go to Professor Dr. Steve MacDonald for carefully reading this work and his helpful comments.

I would also like to thank Ashif Harji, who directly and indirectly contributed to this thesis through his much appreciated advice and technical support, as well as Jiongxiong, Richard and Rudolfo from the PLG lab.

Trademarks

Java is a registered trademark of Sun Microsystems, Inc.

Background

This thesis deals with a specific area of exception handling. Experience with a language using exceptions is therefore helpful for the understanding of this work. Although Chapters 1, 2, and 4 are kept general and should be applicable to any language (provided it offers a suitable EHM), the examples in this work are almost entirely in C++. Therefore, readers should have an understanding of C++, especially of the exception handling facilities. Similarly, to understand some of the specifics of μC++, knowledge of this dialect is helpful. For further information about μC++, visit the following site: http://plg.uwaterloo.ca/~usystem/uC++.html

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This work deals with exception handling. This chapter introduces the basic concepts in exception handling as well as some of the terms used in later chapters. For a more complete overview of exception handling, see [2].

1.1 Overview

1.1.1 Definition of exception

It is hard to define what an exception actually is since there is no widely accepted definition. For the purposes of this work, an exception, in its most basic form, is defined as an event during execution of a program that signals an exceptional situation. An exceptional situation is expected to occur with low frequency. Examples of such situations are division by zero, I/O failure, end of file, or pop from an empty stack. All other situations are handled as part of the general algorithm.

1.1.2 Traditional handling mechanisms

Return codes and status flags

The 'traditional' way of signaling exceptional conditions is with return codes and status flags. The first approach requires a routine to return a value, which often encodes the exceptional condition *and* the normal result of the routine. One example using return codes is the printf() routine from the standard C library, which returns an integer indicating the number of bytes transmitted or (in the case of an error) some negative error code [10].

Status flags use shared variables to indicate an exceptional situation. If such an exceptional situation occurs, the variable is set to some code representing that situation. By checking this variable, it is possible to detect that an exception occurred (until the variable is overwritten due to the next exception). One widely known status flag is the errno variable [10]. These two approaches have numerous disadvantages.

Disadvantages

Primarily, checking of the return value/flag is not mandatory, so it can be delayed or even omitted, which can cause serious errors. Also, exception handling code and algorithmic code are intermixed, which reduces code readability and maintainability. Furthermore, a status flag can be overwritten before the previous condition is checked or handled, and in general, status flags cannot be used in a concurrent environment because of sharing issues.

Non-local handling

Another disadvantage with return values is that exceptions must be checked locally, and hence, are often handled locally. However, local handling is not always desirable since

lower-level code does not need to know the exact reasons for its invocation. To allow non-local handling of exceptions with return values, it is necessary to pass the return code up in the call chain, which is unreliable since every call site has to (re-)check and propagate the return value. However, if a routine calls several routines, all return codes must be folded into the return code for this routine. It is hard to ensure that the routine that is finally capable of handling the condition gets a unique and meaningful result, which sufficiently identifies the exception.

1.1.3 Exception Handling Mechanism (EHM)

Due to the drawbacks of these two approaches, more sophisticated Exception Handling Mechanisms (EHMs) have been developed and are in use today, of which the similar mechanisms for C++ [8] and Java [6] are possibly the most widely known. The events as defined by these EHMs are what most people associate with the term 'exception'.

In the following, the availability of a more sophisticated EHM is assumed.

1.2 Explanation of terms

Event An event is an exception instance as described above. It is generated by executing an operation leading to the signaling of an exceptional situation.

Raise Generating an event is called *raising*. It is also possible to say that the exception is *thrown* and the two terms are used interchangeably in this document[1].

Execution An *execution* is the language unit in which an exception can be raised. For this discussion, exceptions transfer control through dynamic blocks on the stack. An execution is any entity with its own run-time stack, and hence, can support

[1] In μC++, there is a difference between a raise and a throw, but this difference is of a syntactical *not* of a terminological nature.

exceptions. For example, a continuation, a coroutine and a task all have separate stacks.

Handler A *handler* is the part of an execution responsible for reacting to exceptions. One *handler* is usually only responsible for a certain set of exception types. Such a handler is called a *matching handler* for this exception set.

Propagation Propagating an exception directs the control flow of an execution to a handler due to the exception being raised in that execution. This process also involves locating a *matching handler*.

Catch When the *matching handler* for an exception is located and control flow transfers to it, the exception is *caught*. If the handler completes, it is said to have *handled* the exception.

Guarded block A language block with associated handlers responsible for exceptions raised from within this block is called a *guarded block*. An example are try-blocks in C++. The term *try-block* is used as a synonym for *guarded block* in this work.

Synchronous exception When an exception is raised and handled by the same execution, it is called *synchronous*. A *synchronous exception* is propagated immediately after its raise.

Asynchronous exception An *asynchronous exception* is handled in a possibly different execution than the one it is raised in. An execution is not required to propagate an *asynchronous exception* immediately. That is, there can be a delay between the raise by one execution and the propagation in the other execution.

1.3 Properties of an EHM

EHMs are characterized by certain properties. The most important are discussed in this section. This discussion is useful for the understanding of following chapters.

1.3.1 Handling model

Termination

The most common handling model is *termination*. This model is used in most popular languages with exceptions. In the *termination* handling model, all blocks on the run-time stack, located between the raise of the exception and the handler that handles it, are terminated. This process is called *stack unwinding*. Execution continues *after* the block guarded by the handling handler.

Termination is useful in situations in which an exception invalidates the block it is raised in, so continuation of normal execution after the raise is impossible.

Resumption

The other handling model is *resumption*. In this model, control flow transfers from the raise to the handling handler, and after the handler completes, execution continues immediately after the raise point. Unlike *termination*, resumption is non-destructive during handling with respect to blocks on the stack.

The resuming model is useful in situations where it is possible to recover from an exceptional situation and to continue in the block where the raise occurred.

Note that *resumption* can turn into termination (*i.e.*, the stack is unwound) if a handler determines that returning is impossible or the resuming mechanism fails otherwise, *e.g.*, cannot find a matching handler.

1.3.2 Propagation model

The most common propagation model is *dynamic*. This approach searches the dynamic call-stack to find a matching handler for an exception. It also means that matching handlers guarding blocks higher in the call stack[2] have precedence over ones which are lower in the stack.

One of the major consequences of dynamic propagation is that the handler chosen for handling an event is not generally known at compile-time.

1.3.3 Handler clause selection

Like inheritance of classes, inheritance of exception types is a useful facility, providing polymorphism among exceptions. For example, C++ provides inheritance of exceptions. However, inheritance can lead to a situation in which more than one handler for a guarded block matches the raised exception, *e.g.*, if there is a handler for a derived class and a handler for its base class guarding the same block, both can catch an exception of the derived type. Thus, there must be a rule to decide which of the matching handlers is the most *eligible*, and therefore, the one to catch the exception.

To handle this case, most propagation schemes perform a linear search of the handler types, as lexically given by the user, and select the first handler that matches. This approach is both simple for the user and the propagation implementation.

1.3.4 Multiple executions and concurrency

In a concurrent system, it is useful to fold the notion of signals or interrupts into the EHM, which allows exceptions to be sent from a source execution to a target execution, possibly both being executed by different threads. These exceptional situations can be

[2] A stack is assumed to grow up in this work.

handled by *asynchronous exceptions*. After an *asynchronous exception* is raised, it has to be delivered to the target execution. This delivery occurs at certain times which usually cannot be anticipated by the target execution. After the exception is delivered, it is propagated inside the target execution.

1.3.5 Additional important features

Catch-any

It can be beneficial to be able to define a handler that matches any exception. This *catch-any* handler is used when it is necessary to ensure that certain code is executed regardless of what exception occurred. For systems with exception-type inheritance, this functionality can also be emulated by deriving all exception types from a single super-type and defining a handler which is responsible for catching this super-type exception. In systems without a single root for exception types or without exception type inheritance, a special syntax is used for the catch-any handler (*e.g.*, `catch (...)` in C++).

Re-raise

It is useful to be able to catch an exception and for the catching handler to *re-raise* that exact exception. This facility is useful if a handler realizes it is incapable of fully handling the exception, and therefore, it passes the exception onto the handlers further down the stack. This re-raise does not generate a new exception, but rather continues the propagation of the old exception (it is always possible to raise a *new* exception inside a handler).

Notice, that with *catch-any* and exception type inheritance, a handler may not know what exception matched. Therefore, if the handler must continue raising the caught exception, a special re-raise mechanism must exist, either with a special re-raise statement

or with some mechanism to reference the caught exception so it can be raised again.

1.4 Miscellaneous points about exceptions

This section discusses several properties of exceptions that are helpful for the overall understanding of this work or of exception handling.

1.4.1 Exceptions $\neq$ errors

Although it should be clear by the definition given in 1.1.1 that an exception is not an error, many people still make the mistake of equating exceptions with errors. In this discussion, an exception simply signals an exceptional situation, which could be an error, but it could also be absolutely correct and even desirable, as long as it occurs infrequently. EHMs provide alternative control mechanisms that, while especially suited for error handling, could be just as well used for other purposes[3].

1.4.2 Non-local handling

The ability to handle an exception through higher-level code (further down the stack) is a very powerful feature (see also section 1.1.2). However, from a software-engineering standpoint, this feature can be dangerous. When trying to catch low-level exceptions in higher-level code, it can effectively couple the higher-level code to the lower-level implementation.

As an extreme example, it might be possible in a database management system, written in C++, to catch a File_Error inside main(). It may not make sense for main() to know anything about File_Errors, if main() should only deal with general errors. If it does

[3]However, care must be taken when using exception handling in this way since the non-linear control flow – especially with termination – can be quite confusing.

handle the exception, main() is dependent on the implementation of whatever code that raised the File_Error and any changes of that implementation might affect main() as well.

The solution for this problem is to introduce a systematic exception hierarchy (provided derivation of exception types is possible). In this way, it is possible to catch the more abstract exceptions in higher-level code. Upper levels of the exception hierarchy should be less susceptible to changes, so that higher-level code catching these exceptions does not need to be changed. Figure 1.1 shows an example of a common hierarchy structure that could be used for exceptions. For the previous example, File_Error could be derived

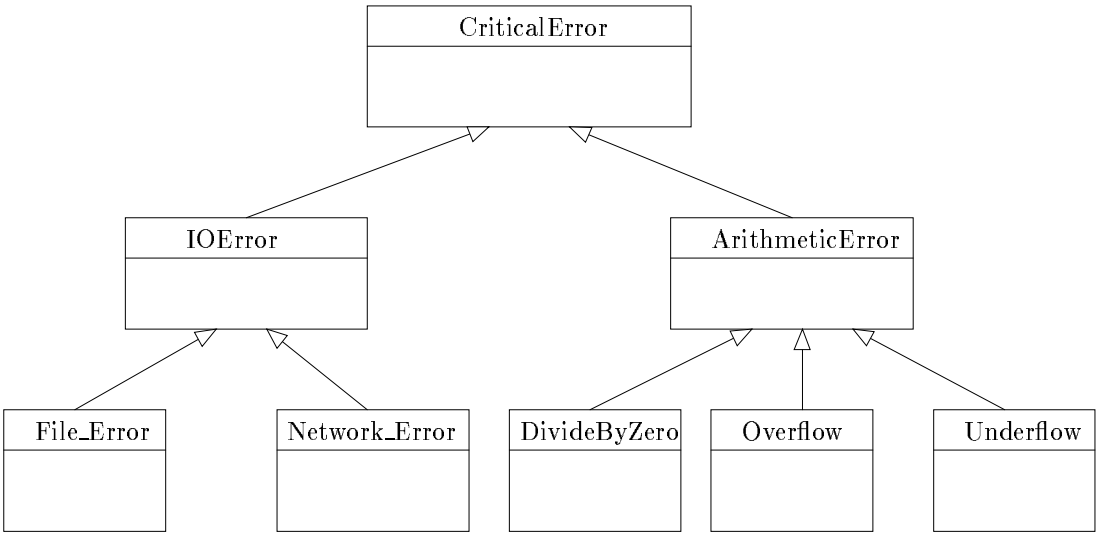

Figure 1.1: Example exception hierarchy

from IOError, and IOError from CriticalError, so that main() can handle CriticalError without establishing unnecessary coupling. It is clear that the higher-level exception handlers cannot deal with the specifics of a certain exception and rather have to resort to some sort of general 'damage control', but this is an inevitable consequence of using non-local error handling.

1.5 Summary

This chapter introduces basic concepts of exception handling. Exceptions indicate exceptional situations. Using an EHM as opposed to traditional error handling methods has many advantages, including code clarity, maintainability and non-local error handling. In this work, an exception is propagated dynamically and can be handled by a *terminating* or *resuming* handler. In an environment with multiple executions, *synchronous* and *asynchronous exceptions* have to be distinguished. If a language supports inheritance of exception types, choosing the right handler to catch an exception is done by selecting the first *eligible* handler found. Finally, there are concepts such as catch-any and re-raise that are helpful features for an EHM.

Building upon this theoretical foundation, it is now possible to introduce the specific concepts that constitute the main focus of this work.

1.6 Thesis outline

Chapter 2 introduces the concept of *Bound Exceptions*. In modern object-oriented programming languages, exception handling does not fit into the overall object-oriented framework since it is impossible to associate exceptions with objects. Bound Exceptions solve this problem. Related concepts are examined and different methods for emulating (statically) Bound Exceptions using conventional EHMs are introduced.

Chapter 3 presents a design for Bound Exceptions. Combining the results from Chapters 1 and 2, Bound Exception functionality is designed for the EHMs of C++ and μC++.

Chapter 4 introduces an advanced feature for Bound Exceptions: *Dynamic Binding*. A motivation for using Dynamic Binding is presented, as well as a method for

emulating Dynamic Binding using statically Bound Exceptions. Finally, these two concepts are compared.

Chapter 5 presents an actual implementation of Bound Exceptions for μC++ using the design elaborated in Chapter 3. The μC++ environment is introduced, as well as the consequences resulting from that environment. Specific problems of implementing Bound Exceptions are discussed and the approaches taken to solve these problems.

Chapter 6 presents the conclusions resulting from this work and provides an outlook to future work that is possible in the area of Bound Exceptions.

Chapter 2

Bound Exceptions

The focus of new work for this thesis is *Bound Exceptions*. This term first appears in [3], but similar concepts have been discussed earlier. In particular, [5] describes an alternative method for exception handling, called *data-oriented* exception handling, for Ada.

This thesis presents a generalized, *object-oriented* approach.

2.1 Basic unbound exception handling

Modern programming languages that support exceptions usually rely on the exception type or the data type of the exception object (in languages which do not have a specific exception type, like C++) to find a matching handler.

Figure 2.1 shows a simple C++ example, in which HANDLER 2 is executed since it is responsible for File_Error exceptions and the raised exception is exactly of that type.

```
class Divide_By_Zero { ... };
class File_Error { ... };
...
try {
    ...
    throw File_Error();
    ...
} catch ( Divide_By_Zero ) {
    // HANDLER 1
} catch ( File_Error e ) {
    // HANDLER 2
}
```

Figure 2.1: Simple C++ exception handling

2.2 Motivation for Bound Exceptions

2.2.1 Limitations of unbound exception handling

The matching-by-type strategy can be insufficient in more complex situations. For example, in Figure 2.2, File_Error exceptions can be thrown by the do_write() method of log_file, data_file and tmp_file. For the matching-by-type strategy, all exceptions are handled by HANDLER A, regardless of which operation actually causes the error (note, Special_File_Error is a subclass of File_Error). This behaviour is undesirable in some circumstances as it is unlikely that errors from three different file objects can be uniformly handled by only one routine. If appropriate, it should be possible to handle each of the three error situations by a separate handler.

2.2.2 Role of exceptions in the object-oriented design

From an object-oriented standpoint, the conventional matching-by-type handling of exceptions is inconsistent. Basically, objects are the main components in an object-oriented software design, and their actions determine the way a program behaves. Hence, an

exceptional situation – as signaled by a raised exception – is the result of an object's action, which suggests that this responsible object should be associated with the exception it raises. Most programming languages do not support this association, leading to a situation in which an exception is treated like something between a type and an object, detached from the overall object-oriented design.

2.2.3 Solution in Ada

Ada [7] is *not* object-oriented and its 'generic packages' are not classes, but they serve as templates from which specific package instances can be obtained. These instances resemble objects closely enough that they are applicable in this discussion.

In Ada, it is possible to bind an exception to specific package instances. Thus, the same exception originating from different instances can be handled separately. In Figure 2.3, the File_Error exception is declared *inside* the generic package Filetype. Filetype has a procedure do_write that raises the exception. The 'pseudo-objects' data_file and log_file are created as instances of Filetype. Hence, it is possible in the exception handler section to distinguish between File_Error exceptions from log_file and those from data_file. Handler A1 is executed in the first case, while A2 is executed in the second case.

It is important to point out that in the Ada approach, it is impossible to handle general, unbound File_Error exceptions any more since File_Error is now part of the generic package and raised exceptions are part of the specific package instance. Although it is possible to raise and handle general unbound exceptions in Ada, package-specific exceptions have to be handled individually for each package instance. This restriction is undesirable as a programmer may want to handle only some cases instance-specifically, while the rest can be handled by one general handler[1].

[1] The fact that most languages with exceptions do not offer any binding capabilities (*i.e.*, handle all exceptions of the same type uniformly), and yet have useful exception handling, supports this desire.

2.2.4 Possible solution in C++

In C++, it would be desirable to write something similar to the Ada example. Figure 2.4 shows a possible solution using an experimental syntax. The *dot-operator* (.) is introduced into the catch clause. The resulting catch clause catch (*object* . *exception-type*) only matches if the thrown exception is of type *exception-type* and bound to *object*. This syntax is unusual as the second operand of the *dot-operator* is a type rather than a field of a structure. Unfortunately, few languages support this kind of exception-object binding for catching exceptions. With languages that lack this feature, such as C++, it is necessary to emulate this functionality to achieve the desired behaviour.

2.3 Emulating Bound Exceptions

2.3.1 Tight Handling

One way to partially emulate Bound Exceptions is to embed each of the three operations in its own try block as illustrated in Figure 2.5. Here, since every method call has its own try block, each error condition can be handled individually. However, because the try block is so tight around the method call, non-local error handling – which is one of the advantages of using exception handling – is impossible. Additionally, since block positioning determines automatic storage allocation and execution control, this example is not logically equivalent to Figure 2.4. If HANDLER A1 catches the exception, the other two blocks are still executed, while in Figure 2.4, execution continues after the single block, and hence, without calling the do_write() member of data_file and tmp_file.

before		*after*
class Err { ... };	→	**class** Err { ... }; **class** CL__Err : **public** Err { ... };
class CL {/* *throws Err* */}	→	**class** CL {/* *now throws CL__Err* */}
CL::foo() { **throw** Err(); }	→	CL::foo() { **throw** CL__Err(); }
catch (Err) { ... }	→	**catch** (CL__Err) { ... }

Table 2.1: Transformations of CL for class-specific exception types

2.3.2 Class-specific exception types

A more complete method to mimic the desired behaviour is through class-specific exception types. Consider the example in Figure 2.6. Here, the Err exceptions are handled by the same handler, even though they originate in different objects of *different classes*. It would be beneficial to be able to distinguish Err exceptions from different classes, while at the same time retaining the ability to handle them all uniformly by a general handler.

Assuming the programming language allows inheritance of exception types, it is possible to derive new exception types CL_Err and LC_Err from Err[2]. Then, every raise of Err inside CL is replaced by a raise of CL_Err, while raises inside LC are changed to throw LC_Err.

By matching these new exception types, the exceptions raised by different classes can be handled separately (*i.e.*, catch (CL_Err) for catching what before was an Err exception thrown inside CL). Table 2.1 summarizes the necessary changes for CL, while Figure 2.7 shows the changes applied to the example from Figure 2.6. Since they are now different types (even though they are basically still the same exception), CL_Err and LC_Err can be handled by different handlers, and because they both inherit from Err, they can both

[2]If the language allows it, the new class-specific type can be defined inside the scope of CL (*i.e.*, CL::Err in C++), which enhances clarity by reducing the number of globally defined classes.

be handled uniformly by a general Err handler.

Class-specific exceptions are supposed to behave like their general type. Therefore, it is necessary to preserve the general inheritance relationships with the new class-specific types. If the general type Err inherits from some general type Superr, then, considering their CL-specific variants, CL_Err must inherit from CL_Superr. Otherwise, catch (CL_Superr) cannot handle CL_Err exceptions, even though catch (Superr) handles Err exceptions. This inheritance requirement can lead to the inheritance structure in Figure 2.8 showing a case of 'diamond-shaped' multiple inheritance, which can be problematic for non-empty classes. For a general discussion of multiple inheritance for exception types, see [2, §6.2].

Implementing the "class-specific exception types" approach, the File_Error example changes as illustrated in Figure 2.9. It is now possible to discern between File_Errors thrown inside tmp_file and those thrown by data_file or log_file. However, in the latter case, it is still impossible to know which of the two Filetype objects actually raises the exception. The problem is that class-specific exception types only allow binding exceptions to classes, not objects.

To overcome this restriction, if the number of instances of a class is known at compile-time, it is necessary to create that many different *object*-specific exception types and associate them with the individual objects at run-time. It is clear that this solution is neither reliable nor practical, given the huge number of classes that would have to be created and that often the number of objects in a program is unknown at compile-time.

Another disadvantage of the "class-specific exception types" approach (apart from its limited binding capabilities) is that it introduces an enormous amount of new classes: the number of classes times the number of exceptions in the worst case.

The major advantage of this approach is that no new program logic or variables are needed. It relies entirely on the existing exception handling mechanism of the language

and no changes to the EHM are necessary.

Finally, correct class-specific exception types are impossible for languages that do not support inheritance of exception types (Ada [7], Modula-3 [12], for example). In that case, the new exception types can still be be created, but without inheriting from the original exception type. While this approach still serves to identify the class an exception is bound to, it is important to realize that the capability to use a general unbound handler is gone. A similar problem exists with the elementary data types in C++. They can be used as exception types, but since they are not classes, they cannot be inherited from.

2.3.3 Catch and re-raise

General outline

A different way to mimic bound exceptions is through the simple "catch and re-raise" approach from [2, §6.4]. If it is possible to pass the 'bound value' from the raise to the catch site, such as the object's id (most likely its address), object and exception are associated, and this association can be interpreted as a binding-relationship. When catching the exception, the passed value can be compared to the desired binding – if they are equal, the exception can be handled, otherwise it is re-raised.

The passing of the value can be achieved via an execution-specific variable or – if the language supports argument/parameter passing during the raise – through the exception itself. In C++, the latter can be done by introducing an attribute into the exception class, as is demonstrated in Figure 2.10.

```
class File_Error { ... };
class Special_File_Error : File_Error { ... };
class Filetype {
    ...
    virtual void do_write() {
        ...
        throw File_Error();
        ...
    }
};
class Special_Filetype : Filetype {
    ...
    virtual void do_write() {
        ...
        throw Special_File_Error();
        ...
    }
};
Filetype log_file;
Filetype data_file;
Special_Filetype tmp_file;

try {
    ...
    log_file.do_write();
    data_file.do_write();
    tmp_file.do_write();
    ...
} catch ( File_Error ) {
    // HANDLER A is always matched
}
```

Figure 2.2: Limitations of matching-by-type

```
procedure BoundExceptions is
    generic package Filetype is
                file_error : exception;
                procedure do_write;
    end Filetype;

    package body Filetype is
        procedure do_write is
        begin
            ...
            raise File_Error;
            ...
        end do_write;
    end Filetype;

    package data_file is new Filetype;
    package log_file is new Filetype;

    begin
        log_file.do_write;
        data_file.do_write;
    exception
        when log_file.file_error  =>  -- HANDLER A1
        when data_file.file_error =>  -- HANDLER A2
end BoundExceptions;
```

Figure 2.3: Solution in Ada – Instance-specific exceptions for generic packages

```
//same definitions/declarations as before
try {
    ...
    log_file.do_write();
    data_file.do_write();
    tmp_file.do_write();
    ...              // Experimental syntax:
} catch ( log_file.File_Error ) {       // bind exception to log_file object
    //HANDLER A1
} catch ( data_file.File_Error ) {      // bind exception to data_file object
    // HANDLER A2
} catch ( tmp_file.Special_File_Error ){ // bind exception to tmp_file object
    // HANDLER 2
} catch( File_Error ) {                 // general handler
    // HANDLER 3
}
```

Figure 2.4: Possible solution in C++ using experimental syntax

```
//same definitions/declarations as before
try {
    log_file.do_write();
} catch ( File_Error ) {
    // HANDLER A1
}
try {
    data_file.do_write();
} catch ( File_Error ) {
    // HANDLER A2
}
try {
    tmp_file.do_write()
} catch ( Special_File_Error ) {
    // HANDLER 2
}
```

Figure 2.5: Tight Handling example

```
class Err{};
class CL {
    void foo() { throw Err(); }
};
class LC {
    void bar() { throw Err(); }
};
...
    CL cl_obj;
    LC lc_obj;
    try {
        cl_obj.foo();
        lc_obj.bar();
    } catch ( Err ) {
        // HANDLER A
    }
...
```

Figure 2.6: Two classes throwing the same exceptions

```
class Err{};
class CL__Err : public Err {};
class LC__Err : public Err {};
class CL {
    void foo() { throw CL__Err(); }
};
class LC {
    void bar() { throw LC__Err(); }
};
...
    CL cl_obj;
    LC lc_obj;
    try {
        cl_obj.foo();
        lc_obj.bar();
    } catch ( CL__Err ) {
        // HANDLER A1
    } catch ( LC__err ) {
        // HANDLER A2
    } catch ( Err ) {
        // HANDLER A
    }
...
```

Figure 2.7: CL example after class-specific transformation

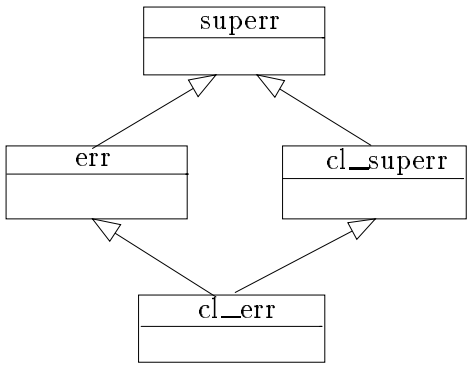

Figure 2.8: Preservation of inheritance structure

```
class Filetype__File_Error : public File_Error { };
class Special_Filetype__File_Error : public File_Error { };
class Special_Filetype__Special_File_Error : public Special_Filetype__File_Error { };

class Filetype {
    ...
    virtual void do_write() {
        ...
        throw Filetype__File_Error();
        ...
        }
};
class Special_Filetype : Filetype {
    ...
    virtual void do_write() {
        ...
        throw Special_Filetype__Special_File_Error();
        ...
        }
};

Filetype log_file;
Filetype data_file;
Special_Filetype tmp_file;

try {   ...
        log_file.do_write();
        data_file.do_write();
        tmp_file.do_write();
        ...
    } catch ( Filetype__File_Error ) {
        // HANDLER A

    } catch ( Special_Filetype__File_Error ) {
        // HANDLER B
    }
```

Figure 2.9: File_Error example after class-specific transformation

```
class Bound_Exception {
  public:
    void * origin;                    // attribute storing the object's ID/address
    Bound_Exception( void * p ) : origin( p ) {}  // store the origin during construction
};

class File_Error : Bound_Exception { ... };
class Special_File_Error : File_Error { ... };
class Filetype {
    ... // somewhere in here, there is a "throw File_Error( this );"
};
class Special_Filetype : Filetype {
    ... // somewhere in here, there is a "throw Special_File_Error( this );"
};

Filetype log_file;
Filetype data_file;
Special_Filetype tmp_file;

try {
    log_file.do_write();          // call can result in "throw File_Error(this);"
    data_file.do_write();         // call can result in "throw File_Error(this);"
    tmp_file.do_write();          // call can result in "throw Special_File_Error(this);"

} catch ( File_Error e ) {
    if (e.origin == &log_file)       // HANDLER A1
    else if (e.origin == &data_file)// HANDLER A2
    else if (e.origin == &tmp_file) // HANDLER B
    else throw;             // re-raise, or handle general File_Error exceptions
    }
```

Figure 2.10: Catch and re-raise

It is clear that this solution is now able to differentiate between exceptions thrown by log_file and those thrown by data_file, which is a major advance over class-specific exception types. On the other hand, this approach increases the program's complexity by adding additional data and code to the exception handling process. In particular, it requires the programmer to follow the strict convention of manually checking the binding information after catching the exception and re-raising it if there is no handler for that binding. Following such a convention is always unreliable. Just one omitted re-throw could cause the program to handle exceptions incorrectly, and the time required for debugging could be substantial.

Re-raise anomaly

Unfortunately, there are situations in which the simple "catch and re-raise" approach does not work (see Figure 2.11). In this example, a Special_File_Error bound to tmp_file is to

```
Special_Filetype tmp_file, dummy;

    try {
        tmp_file.do_write();
        dummy.do_write();
    } catch (Special_File_Error e) {
        if ( e.origin == &tmp_file )  // do something

        else throw;
    } catch ( File_Error e ) {
        if ( e.origin == &tmp_file)  // do something

        else if ( e.origin == &dummy )     // do something else

        else throw;
    }
```

Figure 2.11: Re-raise anomaly

be handled, or a File_Error bound to either object. If tmp_file throws a Special_File_Error

exception, then the first catch clause matches and the handler is executed correctly. However, if it is dummy that throws a Special_File_Error exception, the first catch clause matches but it does not match the binding, and therefore, the exception is re-raised. With normal type matching, if a catch clause does not match an exception, the lexically following catch clauses of the same try-block are checked until a matching one is found (see Section 1.3.3). In the case of a non-matching binding, the re-raised exception cannot be matched against any other catch clause guarding the same try-block. In the example, propagation cannot reach the second catch clause, which would otherwise match (note that a Special_File_Error also is a File_Error) and handle the exception. This kind of anomaly is discussed in [2, §6.4].

The inherent problem of this simple "catch and re-raise" strategy is the following. Without binding, it can be expected that if a matching handler guards a try-block, it catches the exception, while with binding, once the exception is re-raised, it is impossible to catch it with handlers guarding the same try-block. So if the data type of the exception matches but the binding is wrong, it is impossible to match handlers following in the same try-block, and therefore, the exception may not be caught even though a matching (type *and* binding) handler guards the try-block. This behaviour does not match the usual semantics of exception handling, is counter-intuitive, and results in control flow that is difficult to predict.

2.3.4 Enhanced "catch and re-raise"

Preventing the re-raise anomaly

The solution for the re-raise anomaly is to not let related catch clauses be in the same try-block, where 'related' means that there exists an exception-type both of them can catch. A typical example for related catch clauses is one clause that catches some derived

type and one that catches its super-type. When multiple inheritance is possible, catch clauses for all super-types can catch the sub-type. Hence, all super-type catch clauses are related (to the sub-type *and* among each other).

In order to prevent the anomaly, these related catch clauses have to be separated into different (nested) try-blocks, but since the order of catch clauses is important, the catch clauses which lexically follow a related one have to go into the different try-block also. This solution can be formulated into the try-block splitting algorithm presented in Figure 2.12.

start:

start with the first catch clause of the current try block;
repeat
check the type of the current catch clause;
if *a 'related' bound catch clause has been found already in this try-block*:
move the current clause and all following into a new (wrapping) try block;
goto *start;*
continue to next catch clause (if possible);
until *there are no catch clauses to check any more*

Figure 2.12: Algorithm for try-block splitting

Theory of try-block splitting

This try-block splitting algorithm works because the following two constructs

```
try {
    ...
} catch ( Type_1 ) { ... }
  catch ( Type_2 ) { ... }
      ...
  catch ( Type_n ) { ... }
```

and

```
try {
    try {
      ...
    } catch ( Type_1 ) { ... }
      catch ( Type_2 ) { ... }
       ...
      catch ( Type_k ) { ... }

} catch ( Type_{k+1} ) { ... }
  catch ( Type_{k+2} ) { ... }
    ...
  catch ( Type_n ) { ... }
```

are logically equivalent if throws inside the handlers are ignored. The propagation mechanism sequentially checks all handlers, regardless of how many try-blocks are used. Should one handler match, then execution continues at the end of the outermost try-block as there is no code between the two try-blocks. By induction, it is possible to show that this quasi-equivalence holds for any arbitrary number of splittings ($< n$) and resulting try-blocks ($\leq n$).

If there were a perfect equivalence, this approach would be useless since it would just change the code but not the way it works. Now, consider throws inside handlers. In the "catch and re-raise" approach, the problem is that an exception that is caught but rejected, and hence re-raised, is not matched against following handlers. If there is a second try-block enclosing the first one, the handlers of that second try-block can now

catch the exception that would otherwise not be checked, and the anomaly of the simple "catch and re-raise" approach is prevented.

Problems of creating new try-blocks

However, pre-existing throws (*i.e.*, throws which are not part of the "catch and re-raise" implementation) inside handlers now cause a problem. If pre-existing re-throws or throws appear inside one of the handlers, the correctness of the code is not guaranteed as a thrown exception might be handled by a handler in one of the enclosing try-blocks that are created as part of try-block splitting.

To prevent this, pre-existing throws and those of the "catch and re-raise" mechanism have to be distinguished. This distinguishing is accomplished by defining a boolean variable in a new block before the beginning of the original try-block. This variable can then be used to indicate that the current exception is pre-existing. In that case, subsequent handlers can check this variable and ignore the exception (*i.e.*, re-raise it) should they 'accidentally' catch it. After handling the exception, the variable is destroyed by automatic storage management as execution leaves the block.

Conversion example

Figure 2.13 shows a bound exception example using the new experimental syntax and Figure 2.14 shows its conversion following the enhanced "catch and re-raise" approach.

In the conversion, handler D has to be separated into a new try-block since handler C belongs to a bound catch clause of the type Ringo, which is related to D's type Beatle. Handler E has to be put into a new try-block as its type John is related to the bound Beatle of D. The variable OrigThrowPaul acts as a flag to signal that the currently propagated exception is the result of a pre-existing throw. It has to be checked in all subsequent handlers that could potentially catch the re-thrown Paul exception: in this case it has to

```
class Beatle{ };                // exception class definitions
class John : Beatle { };
class Paul : Beatle { };
class George : Beatle { };
class Ringo : Beatle { };

AlbumClass white, pepper;       // just some objects which are going to be bound to
...
    try{
        ...
    } catch ( Beatle ) {  // unbound
        //A
    } catch ( Paul ) {    // unbound
        //B
     throw;             // original RE-THROW
    } catch ( white.Ringo ) { // Ringo bound to white
        //C
    } catch ( pepper.Beatle &e ){ // Beatle bound to pepper
        //D

    } catch ( John ) {    // unbound
        //E
    } catch ( George ) {    // unbound
            //F
    } catch ( white.Beatle ) {    // Beatle bound to white
        //G
    }//try
```

Figure 2.13: Enhanced "catch and re-raise" conversion – before

be checked in G since G handles Beatle exceptions, and therefore, could catch the re-raised Paul exception.

2.4 Summary

Bound Exceptions offer advantages over conventional exception handling. With the enhanced "catch and re-raise" approach, it is possible to mimic Bound Exceptions using conventional exception handling. The conversion process, however, is complicated and

can produce large amounts of additional code. It is unreasonable for a programmer to write this much extra code to replace a missing language feature. Additionally, enforcing such a programming convention – especially if it is as complicated as in this case – cannot be a reliable practice. Therefore, if Bound Exceptions are a desirable feature, it is necessary to implement them as part of the language.

A design of Bound Exceptions as a language feature for C++ and μC++ is presented in the following chapter.

```
{
    bool OrigThrowPaul = false;
    try {
        try {
            try {
                ...
            } catch ( Beatle ) {
                //A
            } catch ( Paul ) {
                //B
                OrigThrowPaul = true; // since this is a user-defined re-throw, set flag
                throw;                // original RE-THROW

            } catch ( Ringo UniqueIdent1 ) {  // first catch the exception, give it a name
                if ( UniqueIdent1.origin == &white ) {
                                              // check if the stored binding is the right one
                                              // if yes, handle exception
                    //C
                } else {
                    throw;            // if not re-raise
                }//if
            }//try
        } catch ( Beatle &e ) {
            if ( e.origin == &pepper ) {
                //D

            } else {
                throw;                // inserted re-raise
            }//if
        }//try
    } catch ( John ) {
        //E
    } catch ( George ) {
        //F
    } catch ( Beatle UniqueIdent1 ) {

        if (OrigThrowPaul )  // check flag in every subsequent eligible handler to re-raise
            throw;           // accidentally caught exceptions

        if ( UniqueIdent1.origin == &white ) {
            //G
        } else {
            throw;
        }//if
    }//try
}
```

Figure 2.14: Enhanced "catch and re-raise" conversion – after

Chapter 3

Designing Bound Exceptions for C++ and μC++

After considering the theoretic properties of Bound Exceptions, this chapter describes an actual design of Bound Exceptions for μC++. The design consists of two part: dealing with the requirements imposed by the base language C++, and then extending the design to the specialties and advanced EHM features of μC++.

3.1 General design for C++

Knowing what Bound Exceptions are and what functionality they should provide, it is now necessary to embed this concept into the target language C++. In particular, it is necessary to decide what types should be Bound Exceptions, where and how they should be defined or declared, as well as when and how the binding occurs and to which object an exception is bound.

3.1.1 Declaration of Bound Exceptions

Following the example of Ada, this section examines if exceptions should be declared as bound and how this should be done.

Bound Exceptions as class members

In Ada, due to the way generic packages work, every Bound Exception is part of the instance it is bound to. Looking for an analogy in C++, defining Bound Exceptions as members of a class would make them part of the objects they are bound to, *e.g.*,

```
class File_Error;        // forward declaration
class Filetype {
    File_Error err;
    ...
};
```

However, for C++, this design is unreasonable. In C++, exception instances are ordinary objects, and as such take up storage space[1]. Hence, every Bound Exception defined as a member of a class would take up storage space for every instance of that class. A list, for example, in which every node object has its own Bound Exceptions could waste a lot of space for exceptions that are probably not needed most of the time.

A related problem arises from the fact that members within classes have to be constructed during instantiation. C++ supports parameter passing for exceptions by storing them as values in the exception objects. If such non-empty exception classes are defined as members, they have to be constructed when their containing object is created. Usually, exceptions are constructed at the time they are thrown, often as anonymous objects.

[1] Even 'empty' objects must have a unique address, and hence, take up space.

In this way, raise-site information is stored inside them, and thus, transmitted to the handling code, *e.g.*,

```
throw File_Error("Disk full");
```

It is also possible to construct the exception object in advance as a named object and to subsequently throw that object, *e.g.*,

```
{
    File_Error err("Disk full");
    ...
    throw err;
}
```

However, early construction of exception objects is uncommon because the data passed into it (*e.g.*, error messages/codes, state information) is usually dependent on the exceptional situation encountered, but this exceptional situation is usually unknown at the time the exception object is constructed. In the above example, a "`Disk full`" situation is anticipated, but the real situation occurring later might be a "`No writing permission`" or a "`Sharing violation`". While it is possible to define member routines inside the exception class to change the stored information, *e.g.*,

```
{
    File_Error err("Disk full");
    ...
    err.changeMessage("Sharing violation");
    throw err;
}
```

this solution is clumsy at best.

By defining exceptions as members within classes, it is either mandatory to construct them before they are used, even though that might not make sense, or to leave them uninitialized during instantiation. In both cases, it is necessary to have special member routines to subsequently insert useful parameter information at the raise. Both of these reasons suggest that exceptions in C++ should not be members of a class.

Note, there are cases in which class members, which usually serve a different purpose, are thrown as exceptions during exceptional situations, *e.g.*,

```
class Filetype {
    string file_name;
    ...
    void fileProblem() {
        throw file_name; // indicate to handler which file has problems
    }
}
```

These cases are not covered in this discussion because the object thrown is usually not used as an exception, and it is just coincidence that it can also be used as an exception.

Bound Exceptions in class scope

C++ also offers the possibility to declare the exception inside the scope of a class. So instead of making the Bound Exception a member of the class, it is possible to declare it inside the class and then access it with the :: operator, *e.g.*,

```
class Filetype {
    class File_Error;  // declaration in class-scope

    void do_write();
    ...
};
class Filetype :: File_Error { ... } ;

void Filetype :: do_write() {
    ...
    throw Filetype :: File_Error("Disk full");
}
```

The advantage of this approach is that there is a lexical association between the exception and the object it is bound to by declaring the exception inside the object class. Exceptions in class-scope also do not take up space or require construction during instantiation.

On the other hand, only being able to use such class-scope exceptions in a bound way is a serious restriction and affects program design (since it is a design decision where to declare classes). This requirement to declare exception classes in class-scope also means that in order to add binding capabilities to old programs (ones that were not intended to use Bound Exceptions) they would have to be substantially rewritten. While catch clauses would have to change (into binding ones), with the class-scope restriction, classes using exceptions would also have to be changed: exceptions would need to be declared inside them. These class changes could affect other parts of the program that would have to be rewritten subsequently.

Also, it is conceivable that binding capabilities could be added to existing code just by (or even without) recompiling it. Of course, this code could not use (*i.e.*, catch) Bound Exceptions as it is not aware of their availability (or the concept), but it could throw them. If programs which are aware of Bound Exceptions use this old code (*e.g.*, as a library), they could catch and handle its exceptions as Bound Exceptions. The following

example illustrates this situation for using the STL vector class.

```
...
using namespace std;
vector<int> v(42);

try {
    v[100] = 0;
} catch ( v.out_of_range ) {
    //handle bound STL exception
}
```

Without changing the source code for the STL, it may be possible to make its classes throw Bound Exceptions. This capability is impossible if Bound Exceptions have to be declared in class-scope since there is no way to change that declaration inside the STL classes (or any other old code) without rewriting it[2].

Bound Exceptions without explicit declaration

Therefore, declaring an exception in class scope is not a good requirement for Bound Exceptions, and hence, the location in which an exception is declared should not affect its binding character. A programmer should decide to declare exceptions inside classes due to the program design, not due to syntactic requirements of the EHM.

3.1.2 Determining the binding character of an exception

Since the declaration location is dismissed for defining the bound characteristic, it is necessary to define what determines whether a particular exception object is bound or not. There are three different methods that could be used here, based on the exception type, the catch, and the raise.

[2]Note that these two legacy issues also apply to Bound Exceptions defined as class members.

Exception type

One way of determining the binding character of an exception is to assign a directive to the definition of the exception type. Thus, all instances of one exception class would be bound and all instances of a different class would be unbound, *e.g.*,

```
class Overflow  { ... };
bound class File_ Error { ... };
```

In the above example, the experimental bound directive is used to declare File_Error objects as bound, while instances of the 'normal' exception class Overflow are implicitly declared as unbound.

This solution is unsuitable for reasons similar to the ones in Section 3.1.1. Again, a class which was not planned to be used as a Bound Exception could never become one because of legacy issues with syntax and semantics. Additionally, the classification of some exceptions classes as bound and others as unbound is inconsistent with the C++ philosophy that 'everything can be an exception'. It is better to give every exception class the possibility to be bound.

Catch

The catch clause has to provide a means to specify how matching is performed for Bound and Unbound Exceptions. Using the experimental syntax again, it is possible to write catch (obj.File_Error) to handle a File_Error bound to obj. At the same time, it should be possible to use catch (File_Error) to handle the unbound version of File_Error. In fact, it makes more sense if the second catch clause also deals with the bound versions of File_Error since those are just special cases of File_Error, and catch (File_Error) suggests that

it provides a handler for *all* objects of that type (see also the discussion in Section 2.2.3).

Raise

It is conceivable to provide a mechanism such that the bound or unbound character of an exception can be determined during the raise. An example would be throw File_Error() for throwing an unbound File_Error exception and boundthrow File_Error() for throwing a bound File_Error exception.

Discussion

Table 3.1 summarizes the four possible combinations of bound/unbound raising and catching and the resulting handling (it is assumed that the exception types match in all cases) when bound and unbound raises and catches are permitted in an EHM.

	(unbound) throw	boundthrow
catch	*unbound*	*unbound*
bound catch	-	*bound*

Table 3.1: Possible options for bound/unbound catching/raising

If the raise is bound and the catch is bound (to the same object[3]), the catch clause is able to provide an object-specific handler, and the exception is handled as a Bound Exception. For a bound raise and an unbound catch, there exists no object-specific handler. The catch clause indicates it can handle *all* exceptions of that type, even the bound ones. Hence, the Bound Exception is handled by a general, unbound handler. If the raise is unbound and the catch is unbound, the case is like conventional exception handling, and the general handler matches.

The most difficult option here is the one for the unbound throw and bound catch. It means that although the user provides a handler for exceptions bound to a specific

[3]The question to what object the exception is actually bound is answered in Section 3.1.3.

object, the exception is still *not* handled since the throw is unbound. This option is of questionable use since higher-level code (lower in the call stack) has no way of handling this exception in a bound way, but rather has to use a general handler, even though it might be able to provide a more object-specific solution. It seems unreasonable to prematurely restrict an exception to unbound handling at the raise since a bound handler should be as well suited (or probably better) to handle it as a general unbound handler.

Therefore, the left column should be dismissed completely, and *all* throws, and hence, *all* exceptions become bound. The decision to handle them as Bound Exceptions or regardless of their binding is made solely in the catch clause. Table 3.2 shows the two remaining options.

	throw
catch	*unbound*
bound catch	*bound*

Table 3.2: Possible options for bound/unbound catching

A positive consequence of this design is that an existing program, which does not know about Bound Exceptions, continues to work after replacing all throws by bound ones. The binding information the exceptions provide is just an additional option that the user can decide to use or not and completely transparent if it is not used.

A negative consequence is that treating all throws as bound wastes resources as it takes memory and time to store binding information which may not be needed. However, the additional space required is marginal compared to the memory usually allocated for normal (unbound) exception handling purposes in current implementations. Also, a raise only occurs in exceptional situations and the performance during exceptional situations is usually slower than the performance during normal situations (see also Section 5.2.3). Therefore, the space and time overhead is not considered to be a problem.

3.1.3 Defining the object binding

It is now necessary to define the object an exception should be bound to. It has already been argued that this object should be the one responsible for raising the exception. It remains to define what the 'responsible' object is. An object's actions take place in its member functions. This suggests that this is also where the exceptions to which it should be bound are raised. Conversely, if an exception is raised inside an object's member function, it should naturally be bound to that object (since all exceptions are bound).

Hence, the bound-to object is defined as the one whose member function contains the raise. For non-member functions (*i.e.*, global functions) as well as static class members, there is no object they belong to – therefore, exceptions raised inside these functions have an invalid binding (which could be indicated by an impossible binding value, *e.g.*, NULL).

As a consequence, if a programmer does not want an exception thrown inside an object's member to have a usable binding, it is possible to define a static member routine for the sole purpose of throwing that exception.

Another consequence is that it is impossible to throw an exception bound to a *different* object (*e.g.*, throw log_file.File_Error). Such a possibility would weaken the object-oriented design of a program and is therefore rejected. To achieve the same functionality, it is possible to define a member function inside Filetype for the sole purpose of throwing a File_Type exception (*e.g.*, log_file.raiseFile_ErrorException();).

3.2 Design for μC++

When trying to extend the design of Bound Exceptions to μC++, it is necessary to consider the advanced exception handling features that μC++ provides. In particular, bound variants for resumption and asynchronous exceptions have to be designed.

3.2.1 Resumption

Introduction

In addition to termination, μC++ provides the concept of resumption. When throwing a terminating exception, the stack is unwound until a matching handler is found; whereas when raising a resuming exception (resumption), the stack is *not* unwound, and after finding and executing the matching handler, execution *returns* and continues immediately after the location of the raise.

Extending Bound Exceptions to Resumption

Extending the concept of Bound Exceptions to resumption is straight-forward since there are no differences with respect to matching between resumptions and exceptions.

3.2.2 Asynchronous Exceptions

Introduction

For both exceptions and resumptions, μC++ provides the concept of asynchronous propagation. Through this feature, it is possible to send events (terminating exceptions or resuming exceptions) from a source to a target task[4], which are delivered asynchronously.

Unsuitability of normal binding

When an event is raised in one task and propagated to another, it may be possible to argue that if there is a responsible object in the source task, the exception should be bound to it according to the rules established in Section 3.1.3. However, it is unlikely that this object is meaningful to the target task. Even if the target 'knows' that object,

[4]More precisely, it is actually possible to send events from/to any coroutine, but the task example is probably the most common case.

it does not mean it knows any of the circumstances leading to the event as they occurred in the execution that raised the exception. Also, the programmer would have to prepare bound handlers (in the target task) for all different kinds of exceptions bound to all different kinds of objects, because unlike synchronous exceptions for which at any given point there is a very limited number of bound exceptions that can be possibly thrown, the target task has no control over the sending execution and events can essentially happen anytime, anywhere[5]

Binding to sender

A better solution is to bind the exception to the sending execution (source)[6]. Analogously to the synchronous case, it appears to the target that it is the source which is 'responsible' for the event, hence, it makes sense to bind the event to the source. Also, asynchronous events serve as inter-task communication, and it is helpful to 'sort' events by their sender, *i.e.*, provide different handlers for the same events sent by different tasks. The number of tasks which can send events to and are known by the target is often limited, so that the number of bound handlers should be small. In cases which have a large number of different tasks communicating with the target (*e.g.*, clients communicating with a server), it is unnecessary and undesirable to provide bound handlers, so that handling of those events is accomplished by a general, unbound handler.

3.3 Summary

This section presents a design of Bound Exceptions for C++ and μC++, which yields the following results. All exceptions are bound, but some can have an invalid binding.

[5]In reality, there are clearly defined times at which events are delivered, but some of them cannot be anticipated by an execution.

[6]Note that in μC++, tasks and coroutines are objects as well.

The decision to handle an exception by a bound or an unbound handler is solely made in the catch clause. A *synchronous* Bound Exception is bound to the object in whose member function it was raised (if applicable). An *asynchronous* Bound Exception is *always* bound to the sending execution. Finally, Bound Resumptions are treated just like Bound Exceptions with respect to handler matching.

The next chapter introduces a different kind of binding strategy: *Dynamic Binding.*

Chapter 4

Dynamic Binding

The method of binding an object/exception pair at the raise discussed so far is called *Static Binding* since the binding information remains unchanged once it has been stored. This chapter introduces *Dynamic Binding*, a new kind of binding strategy, which has significant advantages over Static Binding[1].

4.1 Limitations of static binding

4.1.1 Problems of non-local error handling

Many popular programming languages use an EHM that performs stack unwinding. This capability is a very powerful feature, as it allows 'postponing' the handling of an exception to a lower scope in the call stack and enables non-local error handling. Statically Bound Exceptions, as described so far, have these features, as they behave exactly like their unbound counterparts with regard to stack unwinding. However, the binding information is not as useful after stack unwinding. In fact, the lower down the stack the catch clause

[1] Note that the terms Static Binding and Dynamic Binding in the context of this thesis only refer to the binding of exceptions to objects.

is located (compared to the point where the Bound Exception is raised), the less likely it is that the exception's binding information is meaningful at that point.

This point is illustrated in the following example in Figure 4.1. The first catch clause

```
...
class Filetype;
class File_Error;    // forward declarations

class Database {
    Filetype &f;  // initialized by a constructor
    ...
  public:
    void commit() {
        try {
            f.do_write();  // throws File_Error
        } catch (f.File_Error) {}              // OK
    }
};

void DB_Manager :: flush( Database &db ) {
    try {
        db.commit();
    } catch ( f.File_Error )    {}      // SYNTAX ERROR
    } catch ( db.File_Error )  {}      // OK, but never matched
}


class MainDriver {
  public:
    static void run() {
        DB_Manager dbman;
        Database db( /* construction arguments */ );
        dbman.flush( db );
        ...
    }
}
```

Figure 4.1: Problems of non-local error handling

inside commit() is close to the raise of the exception and represents the typical case of a bound catch.

The first catch clause in flush() (same as for commit()) seems logically possible because the bound object is f. However, it is syntactically impossible since f is not inside the scope of flush().[2] At that point, while the exception object is carrying the binding information for the object f, it has become outdated as the stack is unwound.

Finally, there is the third catch clause: catch (db.File_Error). This clause is probably what a user would like to write, as db exists in the current scope and (from a logical point of view) is responsible for throwing the exception. However, since the exception is bound to f, the handler for this third catch clause is never matched. So, only the the first (close to the throw) catch clause is actually useful for the statically bound case, which shows there are reasonable situations in which static binding is inadequate for non-local error handling.

In general, in order to successfully catch a Bound Exception, the object in the bound catch has to exist in the current scope and has to actually match the object to which the exception is bound.

Coincidental case

Notice, there exists a coincidental possibility that there is an object f in the scope of flush(), *e.g.*, a global Filetype variable f. In this case, there would be no compile-time error, but the bound-to object inside the exception does not have to be this f – in which case the catch clause would be misleading. In fact, once inside flush(), the binding information for f stored in the exception object can be used for bound catching only if this f references a Filetype object visible in flush() and the catch clause tries to bind to that object. In other words, if f references some global Filetype variable called some_global_variable, the catch clause has to be catch (some_global_variable.File_Error).

[2]There is a coincidental possibility that this code is valid if another symbol f is visible inside flush(). This case is discussed shortly.

4.1.2 Software-Engineering considerations

Coupling problem

The requirement to have some_global_variable visible in flush() is problematic and dangerous from a software engineering standpoint, as it prohibits a clean interfacing and encapsulation of code modules. More specifically, by testing for a binding to some_global_variable inside flush(), this code is very strongly coupled to the implementation of the Database class. Even a catch clause much higher up in the call hierarchy would have to know these implementation details. This structure affects maintainability and reuse of the code, and in general, inhibits the use of statically Bound Exceptions in libraries or other situations where there is no access to the code that potentially throws an exception.

Possible Solution

To avoid these problems, Database could provide some kind of interface to reveal the address of f. With this interface, catch (db.getAddress().File_Error) can be used to properly match the Bound Exception. However, to use this strategy consistently, getAddress() methods must be supplied for every object that a class uses - and recursively for every object that these objects use. This requirement would result in an explosion of the number of methods, and even if it were practical, it would reveal the number of objects a class uses - which again is questionable from a software-engineering standpoint.

4.2 Principle of Dynamic Binding

4.2.1 Theoretical considerations

Hypothetical solution

Most static-binding problems could be solved if the exception changed its binding during propagation 'through' an object, *e.g.*, from f to db in Figure 4.1. Naturally, db is inside the scope of flush(), so there would be no compile-time errors. At the same time, flush() would not need to know any implementation details of Database. It suffices to know that Database can throw (or rather propagate) a File_Error exception. For flush(), it would also make more sense to test for binding to an object it uses directly, as opposed to some magical global variable. Thus, all the software-engineering issues listed before are mitigated once the bound-to object is re-associated from f to db. Note also that this solution is compatible with the common sense expectation that if an exception is thrown as a result of a call to db.commit(), then db is actually the object responsible for the throw, and hence, is the one to which the exception should be bound.

Defining "current-level responsible"

In general, in any given try block the exception should be bound to the object whose member function, called in that block, is (possibly indirectly) responsible for raising the exception. This object and the call to its member are called current-level responsible.

4.2.2 Possible Implementations

Catch and re-raise

One way to achieve this binding is to explicitly catch and re-raise all bound exceptions in all methods callable from different objects, as shown in Figure 4.2. Through this

```
...
class Filetype;
class File_Error;                    // forward declarations

class Database {
    Filetype &f;
  public:
    void commit() {
        try {
            f.do_write();    // throws File_Error
        } catch ( File_Error e ) {
            throw e;          // changing bound object through catch and re-raise
        }
    }
};

void DB_Manager :: flush( Database &db ) {
    try {
        db.commit();
    } catch ( db.File_Error )    { // handle error }
    } catch ( File_Error e )  { throw e; }    // or catch and re-raise
}
...
```

Figure 4.2: Catch and re-raise for emulating Dynamic Binding

convention, an exception is always bound to the last object which re-raised it, which is also the current-level responsible object as the binding follows the call stack. In the database example above, the File_Error exception inside flush() would be bound to db, so that catch db.File_Error can be matched.

However, any such programming convention applied by programmers, *i.e.*, to catch and re-raise every exception in virtually every method, is unreliable. Additionally, since every method call has at least one try block and one handler, depending upon the implementation of the EHM used, the increase in exception handling code can result in a significant increase of code size or a slower execution of every method call[3].

[3]See also Section 5.2.3 on page 66.

Proper Dynamic Binding

A better solution is to *automatically* change the binding association during exception propagation by following this convention. For this purpose, a new kind of binding strategy is introduced: Dynamic Binding.

Note that the term dynamic/static binding does not describe whether the binding is done at run-time or compile-time (as the binding always occurs during run-time), but rather determines how rigid the binding of objects to exceptions is.

4.2.3 Dynamic Binding in detail

Dynamic Binding describes the binding strategy in which, unlike static binding, the object to which the exception is bound changes during stack unwinding - just like in the example above, but without explicitly catching and re-raising the exception.

With this binding strategy, non-local handling in flush() finally works as intended by the catch (db.File_Error) clause:

```
void DB_Manager :: flush( Database &db ) {
    try {
        db.commit();
    } catch ( db.File_Error )        // Note the valid binding to db here
}
```

Here, the call of db.commit() is current-level responsible for raising the exception, and consequently (by using Dynamic Binding), the exception object is bound to db. The catch clause expects this binding and correctly matches the exception.

The dynamic nature of the binding has further consequences. If the exception is not handled inside flush(), it is propagated down the stack, and therefore, MainDriver :: run(), which calls flush(), has to be considered. This routine expects an exception to be bound

to the instance dbman of DB_Manager whose flush() member it called and not to db, as ideally, it knows nothing about the implementation of flush() and how db is used inside it.

Algorithm

With dynamic binding, the rules determining the current binding are as described in Figure 4.3.

1. Initially, binding is like static binding
2. For every stack unwinding:
   (a) if there was a current object in the previous stack frame, change binding to that object
   (b) if there was no current object in the previous stack frame, do not change binding

Figure 4.3: Algorithm for Dynamic Binding

Discussion

An interesting problem of the algorithm is to determine what happens if the exception passes through a stack frame without a current object (*e.g.*, a global function, a static member) during propagation. Besides the option chosen (do not change the binding), it is conceivable to change the binding to NULL or another value indicating that there is no binding. This solution is appealing because it is consistent with the rest of the algorithm (*i.e.*, change binding to the current object). However, this approach violates the principle of modularity, which demands that there should be no difference between executing a piece of code in-line and wrapping it into a function and calling that function. The programmer could decide to move a section of the code into a general function. In this case, Bound Exceptions passing through that function during propagation might not

be handled anymore since now, they are unbound, when before they where bound to some object. If the binding information stays the same, the catching code continues to work because nothing changes for the exception.

Using the example in Figure 4.1 without the catch clauses and now utilizing Dynamic Binding, Figure 4.4 illustrates the changing of binding information for the thrown File_Error exception during stack unwinding, following the algorithm from Figure 4.3. Inside f.do_write(), the binding is to f since it is the initial binding (rule 1). After one

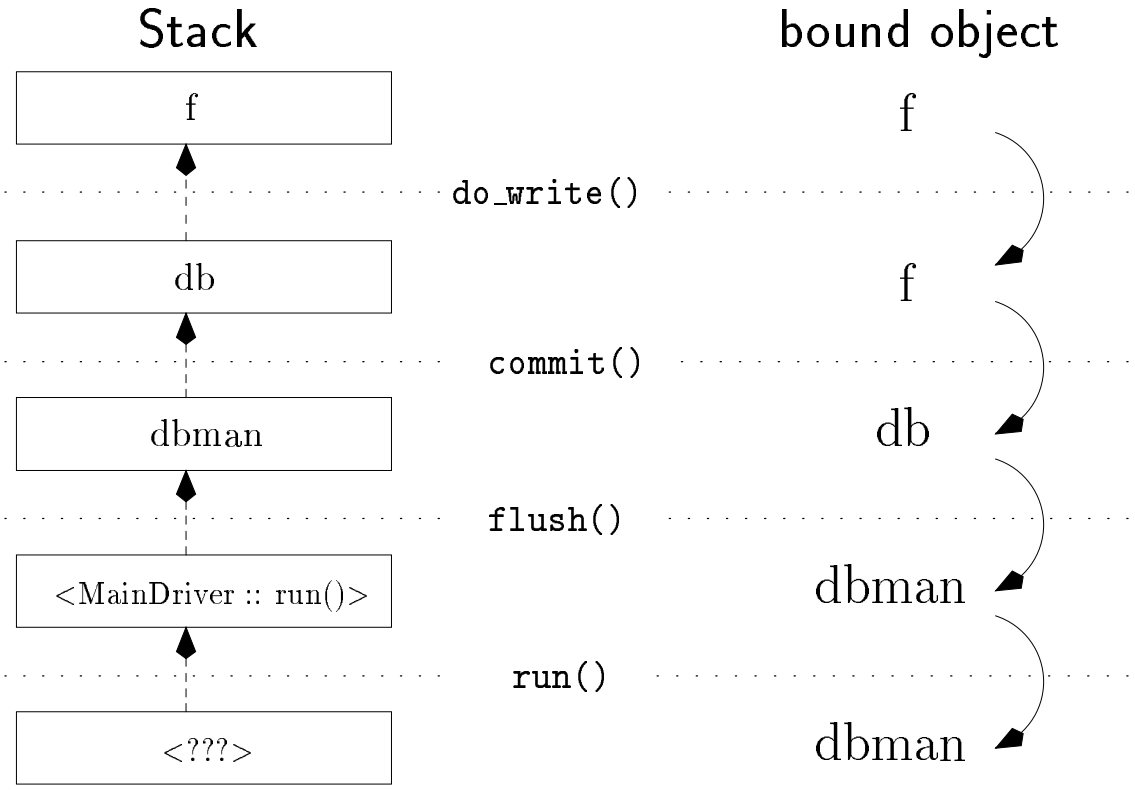

Figure 4.4: Change of binding information

level of unwinding, inside db.commit()'s stack frame, the binding remains to f because the previous stack frame had f as its current object (rule 2a). Note, that up to this point, Dynamic and Static Binding behave identically. During the next level of stack unwinding, the

binding changes, so that inside dbman.flush(), the bound-to object is db according to rule 2a (using Static Binding, the binding would remain f). In MainDriver::run() the binding is to dbman since that was the object of the previous stack frame(rule 2a). Finally, from MainDriver::run() to its caller, the binding remains unchanged because MainDriver::run() is a static member, and hence, there is no current object in that frame (rule 2b).

4.3 Further problems of non-local error handling

Using dynamic binding permits non-local error handling. However, there is a fundamental problem when using (both bound or unbound) exceptions in this way: the data type of the exception always stays the same during stack unwinding. This can lead to a situation in which an exception is (dynamically) bound to an object that at first glance does not have much to do with this exception. In the example from Figure 4.4, a File_Error exception could be bound to dbman, a DB_Manager object – without an obvious association between these two. This situation can be especially confusing if a programmer is used to bound exceptions being declared inside the scope of the bound-to object's class (like for Ada's generic packages). This general problem of exception handling is addressed in Section 1.4.2, and again, a strict exception hierarchy is the solution. Using the example from Figure 1.1 on page 9, it is possible to test for a binding of IOError to db - catch (db.IOError) - and that of CriticalError to dbman, thus mitigating the wrong-association problems.

4.4 Dynamic vs. Static Binding

4.4.1 Discussion of a specific example

Even with the advantages of Dynamic Binding, statically Bound Exceptions are not obsolete. There are situations in which knowing the first bound-to object (*i.e.*, the one

used by Static Binding) is useful. Consider the example in Figure 4.5, which produces the object/call chain in Figure 4.6.

```
void Transactionclass :: do_transaction( Database &db, User_Log &ul) {

    Filetype data_file;
    Filetype log_file;

    try {

        ul.log_events(log_file);
        db.begin_transaction(data_file, log_file);

    } catch ( File_Error ) {}
}

void User_Log :: log_events( Filetype &f ) {
    f.do_write()                        // throws File_ Error
}

void Database :: begin_transaction (Filetype &d, Filetype &l) {
    d.do_write();                       // throws File_ Error
    l.do_write();                       // throws File_ Error
}
```

Figure 4.5: Discussion – Static vs. Dynamic Binding

Advantage of Dynamic Binding

Assuming that log_file's do_write() member throws a File_Error exception which is caught inside Transactionclass :: do_transaction(), it is clear that Static Binding only provides information that the error originated in log_file. However, in order to know if it is the User_Log or the Database object that is responsible for the problem, Dynamic Binding is needed, as it causes the binding information to change to whatever object caused the failing call to log_file.do_write().

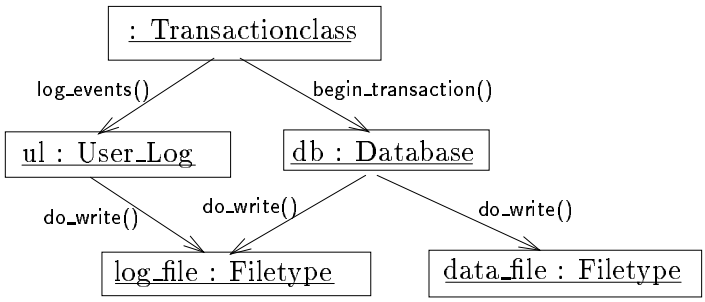

Figure 4.6: Object/Call chain for Transactionclass example

Advantage of Static Binding

On the other hand, if data_file throws a File_Error exception, it is clear that it 'passes through' db, as only db uses data_file, which changes the Dynamic Binding. Once inside do_transaction(), it is impossible to know if this exception originates from data_file or from log_file, as the only information available at that point is a File_Error bound to db. So Dynamic Binding does not help in determining what Filetype object actually has a problem – for this, Static Binding is needed, which stores the original thrower and provides the required information.

Summary

Here is a situation in which the coexistence of both Static and Dynamic Binding is helpful, and since every dynamically Bound Exception behaves like a statically bound one on the highest stack level, there should be no implementation issues that prevent having both forms of binding (it is just necessary to use more storage to remember the first bound-to object). So although Dynamic Binding is generally preferable to the static version, there is a good reason to keep and use both.

4.4.2 Generalization: When to use Static Binding

In general, Static Binding is preferable over Dynamic Binding if there is tight coupling between objects on different levels of the call stack. A good example for this situation is a messenger object that delivers messages from code which is responsible for catching exceptions thrown from the message receivers, as shown in Figure 4.7. Here MainDriver is

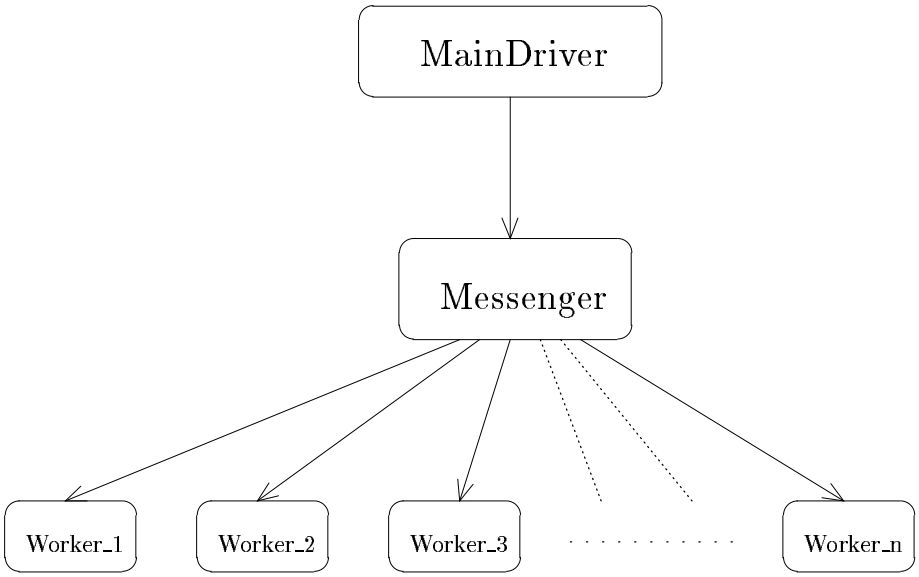

Figure 4.7: Messenger example

not interested in bindings to Messenger but rather in knowing which of the worker objects raised an exception. As they are tightly coupled, MainDriver knows these objects and their implementation sufficiently well so that the exceptions they throw are meaningful to it. Static Binding preserves the original binding, which is useful in this situation.

4.4.3 Further theoretic considerations

Of course, a more complicated object/call tree is conceivable, which would require remembering the objects 'between' the first (static) and the last (dynamic) binding, resulting

in even more complex binding options/strategies. In this case, however, there would be no bound to the number of possible objects 'in between' or the possible different strategies. Therefore, it would be unreasonable to try to add more binding strategies, as all possibilities cannot be covered (and it would be difficult to use). Therefore, only Static and Dynamic Binding are suggested, and by using a good exception handling design (*i.e.*, where to place try blocks and catch clauses), it is possible to cover all the cases in between.

A proposal for the syntax of Static and Dynamic binding can be found in Section 5.3.

4.5 Summary

This chapter introduces *Dynamic Binding*. It showed that handling exceptions non-locally is difficult with Static Binding. Dynamic Binding can replace Static Binding in many cases and enables the combination of Bound Exceptions and non-local handling. Still, it is necessary to use Static Binding in cases when the initial binding must be preserved.

Using the concepts discussed in the previous chapters, the following chapter presents an implementation of Bound Exceptions for μC++.

Chapter 5

Implementing Bound Exceptions for μC++

5.1 Overview of μC++

μC++ is basically a concurrent dialect of C++. It provides high-level concurrency as well as other advanced features as part of the language. The concurrency features of μC++ are provided indirectly by a *translator* and a run-time library, not directly through the compiler. For the actual compilation, μC++ relies on the C++ compiler of the *GNU Compiler Collection* (GCC).

5.1.1 Interaction between μC++ and GCC

To build a program with μC++, the translator first reads in the source file. It then transforms the source from the μC++ dialect into C++ code and calls to the μC++ run-time libraries. The output is then compiled by GCC and finally linked against the μC++ run-time libraries (among others).

Advantages

This design greatly reduces the complexity of the actual μC++ part, and at the same time takes advantage of GCC's optimizing code generation and its availability for many platforms. Otherwise, μC++ could not be made available for as many platforms as it supports now.

Disadvantages

Since μC++ is conceptually only a concurrency library, it suffers from the shortcomings of all such solutions [1]. More importantly for the purpose of this thesis, μC++ has little control over GCC's code generation or its C++ run-time libraries.

5.1.2 Technical view of the μC++ EHM

The μC++ translator and run-time, as well as the GCC run-time library, are all involved in μC++'s EHM. More precisely, μC++ uses its run-time system to handle resumption and asynchronous delivery, The C++ EHM is used for propagating terminating exceptions (stack unwinding, handler matching), which is a combination of compiler and run-time mechanisms.

5.2 Adding binding capabilities to μC++

The dependence on the GCC exception handling mechanism becomes a problem in adding Bound Exceptions to μC++'s EHM. Determining the right handler according to exception type *and* binding information is part of the matching process for which GCC's exception handling routines are responsible – but these routines know nothing about binding.

5.2.1 Extending GCC

One way to solve this problem is to add binding capabilities to GCC's EHM, thereby effectively incoporating Bound Exceptions into the C++ language. This effect would be highly desirable. However, in order to achieve this, it would be necessary to rewrite parts of GCC, especially the run-time system.

Discussion

This solution has several drawbacks. First, GCC is a very complex software project. It would take considerable time and effort to examine how exactly its EHM works, to change the run-time code involved, and to add the new language features to GCC's C++ front-end.

Then, unless the GCC maintaners can be convinced to adopt Bound Exceptions as a standard, one would have to maintain a separate branch of GCC. In time, the main branch and the Bound Exception branch would move farther and farther apart since it is difficult to keep up with the pace of GCC's development. At some point, the Bound Exceptions branch would be out-dated and so would μC++.

Finally, μC++ users would have to install the supplied special version of GCC as well, which could cause conflicts with already installed versions of GCC or system libraries.

5.2.2 Emulating Bound Exceptions

To avoid above difficulties, it is necessary to find a way to implement Bound Exceptions while still relying on the basic C++ EHM. In Section 2.3.4, exactly such a way was developed: the programming convention "enhanced catch and re-raise" emulates Bound Exceptions using a regular non-binding EHM.

While the solution was discarded as too complicated and unreliable to be used by

a programmer, μC++'s translator does not suffer from these limitations. The μC++ translator can use this conversion algorithm to convert from μC++ Bound Exceptions code to plain C++ exception handling code while preserving the binding semantics. This approach is the one used in μC++ to implement Bound Exceptions.

5.2.3 Performance considerations

Naturally, emulating a certain functionality instead of implementing it directly reduces a program's performance. In the following, the performance effects of emulating Bound Exceptions compared to using unbound exceptions are examined. In this particular case, it is necessary to differentiate between performance during normal program execution (non-exceptional case) and exception handling (exceptional case).

Non-exceptional case

The regular, non-exceptional case is examined first. GCC's EHM has two different implementations, one based on the C routines setjmp/longjmp [9] and one based on *DWARF2* information [15]. Without going into too much detail, the *DWARF2* approach, which is the default implementation on the platforms supported by μC++, uses address tables and has the major advantage that exception handling code has no run-time performance impact if no exceptions are handled. It does not matter how many try-blocks or catch clauses a program contains, there is no slow-down as long as no exceptions are actually raised and handled. Hence, the only difference between the unbound and the bound simulation case is slightly larger code size in the latter due to some extra code and bigger address tables (caused by additional try-blocks).

Exceptional case

The case of a raised exception is more complex for the bound simulation case. First, the binding information has to be stored[1], which does not occur for unbound exceptions. The overhead for this is negligible, though.

During propagation, before the exception is caught, there is no difference between bound and unbound handling.

The biggest performance impact occurs during catching. For a single original try-block with n catch clauses, try-block splitting can produce up to n nested try-blocks with one catch clause each in the worst case, which can lead to n catches and re-throws, compared to only one catch in the unbound case. That is quite a significant difference[2], but it is important to realize that this only occurs when actually handling an exception.

Results

The definition of exceptions given in Section 1.1.1 states that exceptions occur infrequently. Hence, performance penalties resulting from emulating Bound Exceptions, which are only experienced in those exceptional situations, also occur infrequently and should have little impact on the overall run-time performance of the program.

In general, a sound philosophy is to not let exception handling code affect non-exception handling performance (as with *DWARF2*), so that the exception handling code will have little influence on the overall performance. See also [13, §16.2 and §16.9.1].

---

[1] How exactly this is done is described in Section 5.4.1

[2] For exact measurements, see Appendix A.

5.2.4 Dynamic Binding

The conversion algorithm from Section 2.3.4 only considers Static Binding. The implementation of Dynamic Binding is inherently more difficult as the binding information changes during propagation, over which there is no control from the μC++ run-time system. It would be necessary to somehow interrupt exception propagation after *every* stack frame unwinding, so control can return to the μC++ run-time, which could change the binding information and then return control to the propagation mechanism again. There is no way to directly interrupt GCC's propagation mechanism, so again, some work-around is needed.

Catch and re-raise approach

In Section 4.2.2, a catch and re-raise[3] convention was introduced that can simulate dynamic binding, provided that static binding is possible.

The reliability issues associated with such a regimen do not apply when using the μC++ translator. However, such a solution would still have the code size problems discussed in Section 4.2.2.

Dummy object approach

A different approach to achieve dynamic binding functionality uses constructors and destructors of dummy objects to change the binding information of an exception during stack unwinding. The technique applied here is called "resource acquisition is initialization" in [14, §14.4]. The following example shows the definition of such a dummy class.

[3]It is no coincidence that "catch and re-raise" approaches are helpful for emulating EHM features. A catch effectively interrupts an exception's propagation, while a re-raise continues that propagation.

```
class DummyClass {
    void * binding;
  public:
    DummyClass( void * pointer ) : binding( pointer ) { }
    ~DummyClass() {
        thisExecution().setExceptionBindingTo( binding );
    }
};
```

The function thisExecution() returns a reference to some object maintaining state information for the current execution. This object is responsible for storing the binding of the current exception. The translator can insert an instance of DummyClass at the beginning of every non-static member function definition, such as in

```
void SomeClass :: someMember() {
        DummyClass _unnamed_( this );     // <== inserted by translator
// **** start of original user code ****
    ...
}
```

If an exception causes the termination of the someClass :: someMember() block during stack unwinding, the DummyClass instance is destroyed and its destructor executed. This destructor sets the current binding to the this pointer that was valid when it was constructed. For the next stack frame, that pointer referenced the current object of the previous stack frame, as required by rule 2a of the Dynamic Binding algorithm from Figure 4.3.

In the catch clause, the binding stored by the previously destroyed dummy object is compared to the one supplied by the user for the catch clause, and if they are equal, the handler matches.

Performance

This solution has the advantage that it just slightly increases code size. It is also faster than the "catch and re-raise" approach discussed before since it does not involve the notoriously slow exception handling engine through repeated throws and catches, but rather executes very few operations to achieve the same effect.

However, it must be clear that by using this approach, regardless whether there is an exceptional situation or not, a dummy object has to be constructed for *every* member function call, and *every* time a member function terminates (stack unwinding, or return), a dummy object has to be destroyed.

Conclusion

Adhering to the policy that exception handling code should have no (or very little) impact on the performance when not actually handling an exception, it was decided to not implement Dynamic Binding in μC++ until a better solution can be found.

5.3 Syntax

The following introduces the syntax for using Bound Exceptions in μC++. The format is partly based on the grammar from [8]. However, it has been extended to reflect semantic requirements. For example, *uDualClass-expression* means that the grammar requires an expression, but to make sense semantically, the expression must be of type uDualClass[4].

[4] All exception types that take advantage of μC++'s advanced exception handling capabilities are implicitly derived from uAEHM::uDualClass, which is the root of the hierarchy of all dual events (*i.e.*, events that can be handled through termination and resumption). uAEHM::uThrowClass is the superclass for all termination-only events, uAEHM::uRaiseClass is the root for the hierarchy of resumption-only events.

5.3.1 Throwing a Bound Exception

The following expressions inside a non-static member function (re-)throw a terminating synchronous or asynchronous exception:

Syntax

uThrow;

uThrow *uDualClass-expression*;

uThrow *uDualClass-expression* uAt *uBaseCoroutine-expression*;

There is no change from the non-binding syntax since now, all throws are bound.

Examples

The first example throws a synchronous File_Error exception bound to the current object.

```
uThrow File_Error( IO_CRITICAL );
```

The second example throws a Special_File_Error exception bound to the current coroutine asynchronously at someTask.

```
uThrow Special_File_Error() uAt someTask;
```

5.3.2 Terminating handler

The following shows the general form of a bound catch clause

Syntax

catch (*lvalue-expression* . *exception-declaration*) *compound-statement*

Important here is the *dot-operator* (.), which connects the bound-to *lvalue* on the left with the exception on the right side. The bound catch clause works for any *lvalue*, but for the binding to make sense it should be any object whose member functions can raise exceptions, or any coroutine or task objects raising asynchronous events.

Examples

The following handles all File_Error exceptions bound to the object named obj.

```
catch ( obj.File_Error e)
```

Note the optional *declarator* e, which is used to name the exception object inside the handler.

The following catches a File_Error exception bound to whatever object func_returning_pointer() references.

```
catch ( *func_returning_pointer().File_Error )
```

Note that the expression *func_returning_pointer() is evaluated during *handler matching* even if the handler does not match. Caution is required when using functions in this way, or otherwise unexpected side-effects can occur.

The next example handles all asynchronous exceptions sent by the task someTask

```
catch ( someTask.uAEHM::uThrowClass )
```

Finally, here is an example of a general unbound handler, which accepts all binding information. Again, a *declarator* is supplied.

```
catch ( Special_File_Error &e )
```

5.3.3 Raising a Bound Exception (Resumption)

The syntax for the resuming raise is very similar to that of the throw, the only difference being the use of uRaise instead of uThrow.

Syntax

uRaise;

uRaise *uDualClass-expression*;

uRaise *uDualClass-expression* uAt *uBaseCoroutine-expression*;

Examples

A simple raise of a File_Error exception bound to the current object (this time returning):

```
uRaise File_Error();
```

The following example shows how a task can asynchronously raise an exception and send it to itself[5]:

```
uRaise Special_File_Error() uAt uThisTask();
```

[5]Note that the exception is propagated only after it has been delivered asynchronously. In the previous example, the exception is propagated immediately.

5.3.4 Resuming handler

The μC++ syntax for resuming handlers is very different from the one for terminating handlers (*i.e.*, catch clause). The reason is that a resuming handler is like a nested routine with dynamic name look-up. However, C++ has no provision for nested routines, so an unusual syntax has to be adopted. For more information on how to use resumption, see [11] and [4].

Syntax

try <*lvalue-expression* . *uDualClass-type-specifier* > *compound-statement*

try <*lvalue-expression* . *uDualClass-type-specifier* , *handler-function* > *compound-statement*

try < *lvalue-expression* . *uDualClass-type-specifier* , *handler-function* , *uClosure-name* > *compound-statement*

Again, the difference between the bound and the unbound version is the *dot-operator*.

Examples

The following catches a File_Error exception bound to obj, and it should be handled by the default handler[6] for this type.

```
try < obj.File_ Error > {
    ...
}
```

The following example states that all raised events bound to obj should be handled by foo() with closure containing args

[6] If an event is raised but no handler is supplied, a *default resuming handler* uDefaultResume() is executed. The pre-defined handler throws the event, but it is possible to override it by a user-defined handler.

```
try < obj.uAEHM::uRaiseClass, foo, args > {
    ...
}
```

Finally, the following lets bar() handle all File_Error resumption events sent to the current coroutine by someTask.

```
try < someTask.File_Error, bar > {
    ...
}
```

5.3.5 Dynamic Binding

Although Dynamic Binding is currently not supported by μC++, a possible future syntax should be considered. Like the *dot-operator* for Static Binding, the *slash-operator* (/) could be used for Dynamic Binding.

Experimental syntax

catch (*exception-declaration* / *lvalue-expression*) *compound-statement*

The following line would mean to catch all File_Error exceptions dynamically bound to obj.

```
catch ( File_Error / obj )
```

The inverted order of exception and bound-to object allows using Static and Dynamic Binding at the same time, *e.g.*,

```
catch ( log_file . File_Error / db )
```

Here, all File_Error exceptions statically bound to log_file and dynamically bound to db are handled (see the example from Section 4.4.1).

5.4 Implementation details

This section describes those parts of the EHM in greater detail that are crucial for the implementation of Bound Exceptions.

5.4.1 uThrow

In Section 3.1.2, it was postulated that all exception types should be binding. This raises the question of where the binding information is actually stored since not all types provide a data field for this purpose (especially not the elementary types like int, float, etc.)

Storing binding information as execution-specific data

One approach is to store the binding information in the structure maintaining state information for an execution, like in the dummy approach from Section 5.2.4. Since, at any given time, at most one exception is being propagated in an execution, this approach is safe.

Storing binding information inside exception object

In μC++, in order to use advanced features like resumption or asynchronous exceptions, it is necessary to derive an exception type from uAEHM::uDualClass. As a consequence, all exception types effectively inherit (protectedly) from uAEHM::uDualClass. Hence, to make sure all exception types are potentially binding, it is only necessary to include an

attribute in uAEHM::uDualClass for storing the binding information. This approach is the one implemented in μC++ since, from a design standpoint, it is better for an exception to maintain its own binding information.

The following shows the important parts of the definition of uAEHM::uDualClass (const declarations omitted).

```
class uAEHM::uDualClass {
    ...
    void * uStaticallyBoundObject;
    ...
  public:
    void * uGetOriginalThrower() { return uStaticallyBoundObject; }

    uDualClass * uSetOriginalThrower( void * p ) {
        uStaticallyBoundObject = p;
        return this;
    }
};
```

During the throw, uAEHM::uDualClass::uSetOriginalThrower() is used to store the current this pointer inside the exception. When the translator encounters an expression such as

```
void Mary::foo() {
    ...
    uThrow File_Error();
    ...
}
```

it transforms it into

```
void Mary::foo() {
    ...
    throw *File_Error().uSetOriginalThrower( this );
    ...
}
```

which constructs an anonymous exception object, stores the current this pointer in that object, and then de-references the address returned by uSetOriginalThrower() to throw the exception. Since on the right side of an initial (not re-throw) uThrow there has to be an expression which implements uAEHM::uDualClass, it is clear that this conversion works for any kind of throw (*e.g.*, also if an already constructed, named object is thrown)

5.4.2 uRaise

The transformations done for a uRaise are identical to those of uThrow.

5.4.3 Asynchronous uRaise/uThrow

This part is very similar to the simple uThrow. The only difference is that &uThisCoroutine()[7] is stored inside the event object, and not this.

5.4.4 Resuming handler

Propagation and handler matching for resumption is done entirely inside the μC++ runtime. When a resuming event is raised, the existing resumption clauses (which are stored in a special data structure) are searched linearly (down the stack) until one is found whose type *and* binding match. The handler determined by the found resumption clause is then executed.

5.4.5 Terminating handler

As discussed in Section 5.2.2, the "enhanced catch and re-raise" algorithm is used to implement the matching of terminating Bound Exceptions. So whenever the translator encounters a bound catch clause, it basically transforms it according to the algorithm.

[7] uThisCoroutine() returns a reference to the object representing the current coroutine/execution.

See Appendix B for a comprehensive conversion example. There are three subtle points in which the implementation differs from the algorithm.

Optimization

There are cases in which it is possible to minimize the number of try-block splittings. Consider the following example.

```
Filetype log_file, data_file;
try {
    ...
} catch ( log_file.File_Error e ) { /* A */ }
  catch ( data_file.File_Error e ) { /* B */ }
```

Since both catch clauses are of a related type, according to the original algorithm, they would have to be separated, *i.e.*,

```
try {
    try {
        ...
    } catch ( File_Error e ) {
        /* if bound to log_file, execute A
        else reraise */
    }
 } catch ( File_Error e ) {
    /* if bound to data_file, execute B
        else reraise */
 }
```

However, since the exception types are equal (as well as the declarators) and follow each other, it is possible to achieve the same functionality with only one try-block and catch clause. The following shows the equivalent optimized code.

```
try {
    ...
} catch ( File_Error e ) {
    /* if bound to log_file, execute A
       else */
    /* if bound to data_file, execute B
       else reraise */
}
```

As it is highly probable that bound catch clauses of the same type follow each other and have the same declarator, this optimization should significantly reduce the performance overhead caused by emulation.

Pre-existing throw flag problem

In Section 2.3.4, the solution offered for the pre-existing throws problem is to set a flag for this exception and check it in all subsequent *eligible* (*i.e.*, capable of catching the exception) handlers. In reality, the μC++ translator cannot always determine the *eligible* handlers. If, for example, a routine contains a re-throw and exception handlers call this routine, the type of the re-thrown exception can only be determined at run-time. Additionally, when encountering the routine's definition, the translator does not know which handlers call it. Therefore, in the actual implementation of the conversion algorithm, a single flag is defined before *every* converted try-block, and the flag is checked in *every* catch clause of a transformed try-block.

Multiple Inheritance problem

In the original algorithm, the exception is handled if the address stored in the exception is the same as the address of the object supplied by the user in the catch clause. Since multiple inheritance is possible in C++/μC++, a special adjustment has to be made to

the algorithm[8]

Consider the example in Figure 5.1. Without the adjustment, the Busy exception

```
...
class Teacher {
  public:
    void teach() {
        ...
        uThrow Busy();
    }
};

class Researcher {
  public:
    Paper * research() {
        ...
        uThrow Busy();
    }
};

class Professor : public Teacher, public Researcher {
  public:
    void supervise() { ... }
};

void schedule() {
    Professor Bob;
    try {
        Paper *p = Bob.research();
    } catch ( Bob . Busy ) {
        // Handler
    }
}
```

Figure 5.1: Multiple inheritance problem

would *not* be caught in the example above, for the following reason. Since Professor inherits from both Teacher and Researcher, Bob internally consists of two parts – a

[8]This adjustment applies to GCC versions up to 2.96, which are used in μC++ 4.9. For later versions of μC++, this adjustment is not necessary since the GCC versions used show a different behaviour with regards to this pointers.

Researcher part and a Teacher part (see Figure 5.2). These parts behave like individual

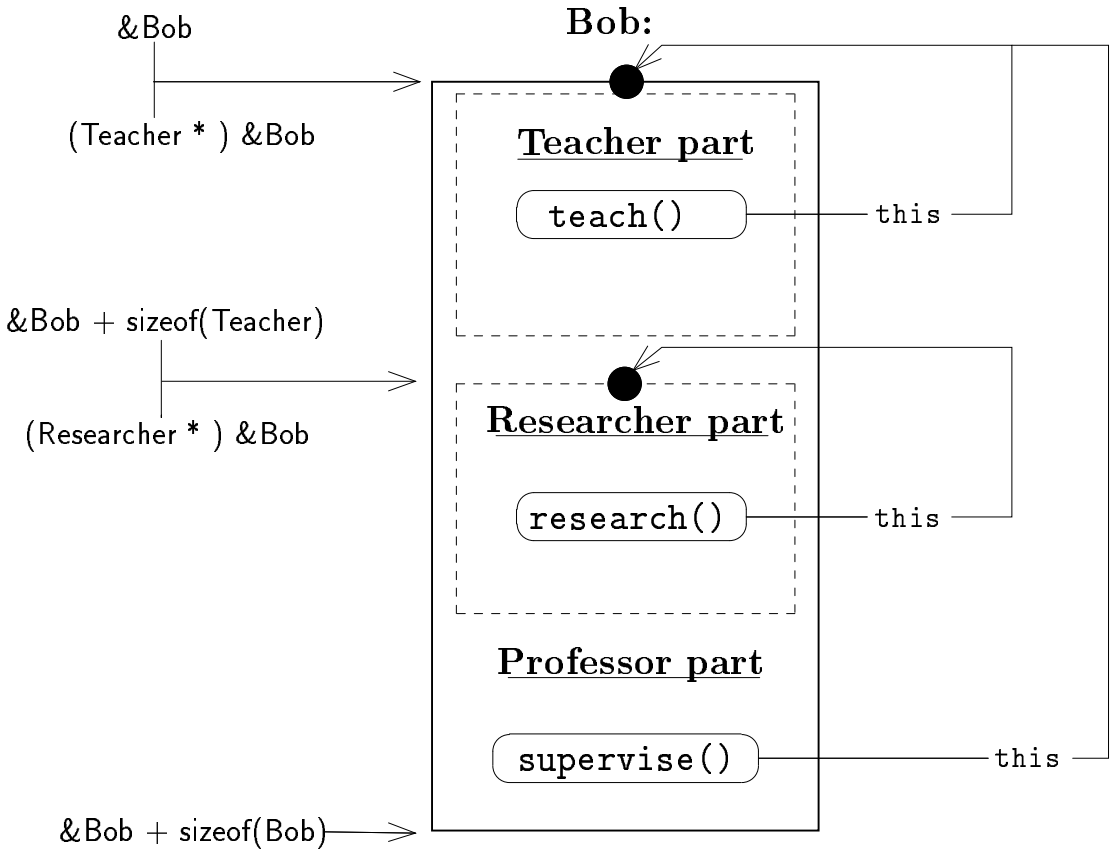

Figure 5.2: Organization of a Professor instance

sub-objects. In particular, calls to the member functions teach() and research() supply *different* this pointers, and hence, throws inside these functions are bound to different addresses. Teacher is the first class Professor inherits from. Therefore, GCC places the Teacher part at the beginning of Bob, so the this pointer for Bob.teach(), which points to the Teacher part of Bob, is the same as the address of Bob. The Researcher part of Bob is placed *after* the Teacher part. Hence, Bob.research(), which belongs to that Researcher part, has a this pointer with a *higher* address (&Bob + sizeof(Teacher)), which is why the binding inside the catch clause fails.

To overcome this problem, the actual matching algorithm checks whether the stored address lies in the range [&Bob , &Bob + sizeof(Bob)), which covers the whole Bob object and therefore also accepts the binding information from Bob.research().

This solution has one interesting consequence. Consider the example in Figure 5.3. Since [&t , &t + sizeof(t)) only covers the Teacher part of Bob, the binding fails again. In this case, however, this is the correct behaviour because a Researcher member function should not be able to bind an exception to a Teacher object, only to a Researcher or Professor object.

```
void schedule() {
    Professor Bob;
    Teacher &t = Bob;    // sub-type assignment
    try {
        Paper *p = Bob.research();
    } catch ( t . Busy ) {
          // Handler
    }
}
```

Figure 5.3: Attempted binding to Teacher part

5.5 Summary

In μC++, a translator is used to implement statically Bound Exceptions. Dynamic Binding is currently not implemented as it would result in significant performance loss or code size increase; the correct way to implement Dynamic Binding is through the compiler, which is beyond the scope of this thesis. In this implementation of Static Binding, the address of the bound object is stored as a field inside uAEHM::uDualClass, which all μC++ exceptions inherit from. When the μC++ translator encounters a raise using Bound Exceptions, it transforms it into plain C++ code that stores the binding information inside

the exception object. In a bound catch or resumption clause, the *dot-operator* connects the bound object on the left with the exception on the right side. For resumption, handler matching is done in the μC++ run-time. Terminating catch clauses are matched by GCC's EHM, and therefore, the μC++ translator converts bound catch clauses using a variant of the "enhanced catch and re-raise" approach to emulate Bound Exception handler matching.

Chapter 6

Conclusion and Outlook

6.1 Conclusion

With the introduction of Bound Exceptions, it is possible to truly incorporate exception handling into the object-oriented design of a program. In particular, I developed the concept of Dynamic Binding, which offers software-engineering advantages over existing binding schemes. The ability to associate exceptions with objects strengthens the relationship between an exception and the object responsible for its raise. This feature creates more powerful exception handling capabilities, contributing to building more robust software.

This work presents the design of Bound Exceptions for C++ and μC++. The benefits of Bound Exceptions are not restricted to these languages alone. Following the C++ example, it should be simple to port the design to other object-oriented languages, as long as they provide a suitable EHM.

The implementation of Bound Exceptions for the EHM of μC++ provides a platform for using this feature under practical circumstances. With resumption and asynchronous exceptions, μC++ supports basically all exception models in use today. Therefore, it

can serve as a test platform that developers can use to evaluate the usefulness of Bound Exceptions in their programs and decide if they are a desirable feature for other languages.

6.2 Outlook

The following lists several future developments necessary or possible in the area of Bound Exceptions.

6.2.1 Field Testing

This work clearly demonstrates the theoretic advantages of using Bound Exceptions. However, to actually prove its practical usefulness, theoretic considerations and 'toy examples' are insufficient.

With μC++, there now exists a tool to create larger, real-world applications exploiting the capabilities of Bound Exceptions. By doing so, it is possible to prove that Bound Exceptions deliver the advantages they promise theoretically.

6.2.2 Conditional Handling

In [2, §6.4], it is shown that Bound Exceptions are a special form of a broader concept: *Conditional Handling*. Through Conditional Handling, an exception is caught by a certain handler only if a supplied condition is met (*i.e.*, the expression is `true`). With Bound Exceptions, this condition is the equality of user-supplied binding and actual bound-to object. Buhr *et al.* argue that there is no evidence for the usefulness of Conditional Handling, but this could be due to the lack of practical testing. It would be very simple to incorporate Conditional Handling into μC++'s EHM since the concept and the required transformations are similar to those of Bound Exceptions.

Possible application

The following shows a possible syntax using square brackets to encompass the condition

`catch ( [` *expression* `]` *exception-declaration* `)` *compound-statement*

Here is an example utilizing *Conditional Handling*:

```
try {
...
} catch ( [ log_file.getLogStatus() == DB_INTACT ] File_Error e ) {
    db.rollback();
    log_file.notify( e.getErrorMsg() );

} catch ( CriticalError ) {
    cleanUpAndAbort();
}
```

If a File_Error is thrown inside the try-block and log_file.getLogStatus() returns DB_INTACT, the handler matches and handles the exception. If the condition is false, the exception is matched to the general CriticalError handler (this happens naturally if the exception is a CriticalError but not a File_Error).

6.2.3 Dynamic Binding

In Chapter 4.4, the concept of Dynamic Binding is developed and shown to be superior to Static Binding in many contexts. It would be desirable to have Dynamic Binding capabilities in μC++ so that their usefulness can be evaluated in practice.

In order to achieve this, it is necessary to find a more efficient way to emulate Dynamic Binding or to accept the performance issues associated with the dummy object algorithm from section 5.2.4. A third possibility is to extend GCC by adding (dynamic) binding capabilities.

6.2.4 Extending GCC

In many ways, the best solution would be to incorporate Bound Exceptions directly into GCC. Given that the problems described in section 5.2.1 can be solved (by convincing the GCC maintainers of the advantages of Bound Exceptions, for example) or are ignored, it would be highly desirable to have GCC recognize Bound Exceptions. Programmers who are uninterested in concurrency would not have to use μC++ in order to benefit from Bound Exceptions, but could just use plain GCC C++. Additionally, all performance issues associated with emulation would disappear, and in particular, it would be possible to implement Dynamic Binding properly.

Appendix A

Performance measurements

In μC++, an emulation ("enhanced catch and re-raise") has to be used in order to implement Bound Exceptions[1]. This emulation cannot be as efficient as a direct implementation. In order to measure the performance overhead of the emulation, two programs are presented here. Section A.1 shows a μC++ program (BEperf.cc) utilizing Bound Exceptions. Section A.2 shows an equivalent program (BEmimic.cc) that, instead of using the μC++ EHM, mimics Bound Exceptions functionality through object-specific exception types (see section 2.3.2). This implementation is very fast since it introduces no new try-blocks or catch clauses.

In both programs, the outer loop is executed REP times, each iteration throwing one exception. The exceptions are evenly distributed over the five catch clauses, so that on average, the propagation mechanism has to check $2\frac{1}{2}$ handlers before the matching one is found. The inner loop is used to simulate 'realistic' algorithmic code by executing $2 \times$ REPREAL arithmetic and $1 \times$ REPREAL I/O operations for every exception thrown. The time for executing the outer loop is measured.

[1] For a comprehensive example explaining how exactly the μC++ translator transforms source code into code that emulates Bound Exceptions, see appendix B.

Both programs are built by executing `u++ BEperf.cc` and `u++ BEmimic.cc`, respectively. By defining PURE during compilation (i.e., `u++ -DPURE`), it is possible to only execute exception handling code, i.e., eliminate the inner loop. For BEperf.cc, it is possible to enable optimization (see section 5.4.5) by defining OPTIMIZE during compilation.

Table A.1 summarizes the performance measurements. All programs were executed 9 times, the median was taken as the measurement. The results when PURE is defined are

	Mimicking	μC++ (OPTIMIZE)	μC++ (no OPTIMIZE)
PURE	$0.93s$	$1.79s$	$3.92s$
Realistic code	$9.12s$	$10.09s$	$12.05s$

Table A.1: Measured execution times

significantly smaller as only exception handling code and no algorithmic code has to be executed. Hence, the effects of the different exception handling mechanisms are reflected in the top row. With OPTIMIZE defined, all bound exception clauses have the same type and declarator. Therefore, it is possible to use only one try-block and one catch clause. Without optimization, five try-blocks with one catch clause each are created, and $2\frac{1}{2}$ catches and (re-)throws are executed on average. These additional exception handling operations are responsible for the longer run-time of the unoptimized program. There are some operations (e.g., setting of pre-existing throws flag, storing of the binding information) that are executed only once in both cases, which is why the run-time of the unoptimized program is less than $2\frac{1}{2}$ as long as that of the optimized program.

Since there is only one try-block with one catch clause in the optimized case, it could be as fast as the mimicking version, which has the same number of try-blocks. In reality, the μC++ code takes about twice as long. This effect can be explained by the additional operations that have to be executed during the raise (i.e., storing of the binding). In fact, if the throws are replaced with uThrows in the mimicking version, the measured time of $1.92s$ is actually slightly higher than that of the optimized μC++ version. The difference

can be explained by the slightly more complex code for choosing the right raise (inside mary::foo()) and that the mimicking version still has five catch clauses, which results in $2\frac{1}{2}$ catch clause checks (on average) by GCC's EHM during propagation.

The bottom row shows that the difference among the Bound Exceptions implementations quickly diminishes when even a small amount of algorithmic code is executed for each exception raise. Therefore, only when Bound Exceptions comprise the majority of control flow is performance an issue. However, it was made clear in section 1.1.1 that exceptions are supposed to indicate rare situations. Hence, when exceptions are used correctly, performance issues resulting from μC++'s emulation of Bound Exceptions are negligible.

A.1 Bound Exceptions using μC++

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

#define REP 10000
#define REPREAL 100

#ifndef PURE
  #define CLOSE f.close();
#else
  #define CLOSE
#endif

#ifdef OPTIMIZE
 #define A
 #define B
 #define C
 #define D
 #define E
#else
 #define A aa
 #define B bb
 #define C cc
 #define D dd
 #define E ee
#endif

uThrowEvent fred { };
uInitEvent( fred );

class mary {
  public:
    void foo( int k ) {
     uThrow fred();
    }//foo
};

void uMain::main () {
   mary a,b,c,d,e;
   uTime start = uThisProcessor().uGetClock().uGetTime();

    for ( int j = 0; j < REP; j++ ) {
#ifndef PURE
     int k = 100;
     uIFStream f("a.out");
     char dummy;
#endif
```

```
      try {
          for ( int i = 0; i < REPREAL; i++ ) {
#ifndef PURE
            k*=21;
            k/=19;
            f >> dummy;
#endif
          }//for
          switch ( j % 5 ) {
            case 0:  a.foo(j);
            case 1:  b.foo(j);
            case 2:  c.foo(j);
            case 3:  d.foo(j);
            case 4:  e.foo(j);
          }//switch

      } catch ( a.fred A) { CLOSE }
        catch ( b.fred B) { CLOSE }
        catch ( c.fred C) { CLOSE }
        catch ( d.fred D) { CLOSE }
        catch ( e.fred E) { CLOSE
         }//try
    }//for

    uDuration dur =  uThisProcessor().uGetClock().uGetTime() - start;
    uCerr << dur << endl;
}//main
```

A.2 Mimicking Bound Exceptions with C++ code

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

#define REP 10000
#define REPREAL 100
#ifndef PURE
  #define CLOSE f.close();
#else
  #define CLOSE
#endif

class fred { };

class fred_a : public fred { };
class fred_b : public fred { };
class fred_c : public fred { };
class fred_d : public fred { };
class fred_e : public fred { };

class mary {
  public:
    void foo( int k ) {
     switch ( k % 5 ) {
       case 0: throw fred_a(); //a
       case 1: throw fred_b(); //b
       case 2: throw fred_c(); //c
       case 3: throw fred_d(); //d
       case 4: throw fred_e(); //e
     }//switch
    }//foo
};

void uMain::main () {
   mary a,b,c,d,e;
   uTime start = uThisProcessor().uGetClock().uGetTime();

    for ( int j = 0; j < REP; j++ ) {
#ifndef PURE
     int k = 100;
     uIFStream f("a.out");
     char dummy;
#endif
     try {
         for ( int i = 0; i < REPREAL; i++ ) {
#ifndef PURE
          k*=21;
```

```
            k/=19;
            f >> dummy;
#endif
          }//for
          switch ( j % 5 ) {
            case 0:  a.foo(j);
            case 1:  b.foo(j);
            case 2:  c.foo(j);
            case 3:  d.foo(j);
            case 4:  e.foo(j);
          }//switch

       } catch ( fred_a ) { CLOSE }
         catch ( fred_b ) { CLOSE }
         catch ( fred_c ) { CLOSE }
         catch ( fred_d ) { CLOSE }
         catch ( fred_e ) { CLOSE;
         }//try
    }//for

    uDuration dur =  uThisProcessor().uGetClock().uGetTime() - start;
    uCerr << dur << endl;
}//main
```

Appendix B

A comprehensive transformation example

The example in section B.1 illustrates the transformation the μC++ translator performs to emulate Bound Exceptions through the "enhanced catch and re-raise" approach (with modifications). It is a purely synthetic example with the sole purpose of demonstrating the criteria used by the μC++ translator in order to decide where to split try-blocks and where to optimize. For each important part of the source code, there is an explanation as a comment *inside* the code itself. Section B.2 shows the output of the μC++ translator for the example. Comments were restored (as they are usually removed during translation) and the output re-formatted so it can be used in this document.

B.1 Example before transformation

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

uThrowEvent Grandmother { };
uInitEvent( Grandmother );

uThrowEvent Father : public Grandmother { };
uInitEvent( Father );

uThrowEvent Aunt : public Grandmother { };
uInitEvent( Aunt );

uThrowEvent Child : public Father { };
uInitEvent( Child );


class someObject {

  public:
    void doSomething() { uThrow Grandmother(); }

};

void uMain::main() {

    someObject fred, mary, john;

    try {
     try {
         // after transformation, notice the second variable
         // uOrigRethrow in this scope overlapping the first one

         mary.doSomething();

     } catch( Aunt ) {
         // unbound Handler, note: uOrigRethrow is checked everywhere
         // after transformation
     }

    } catch ( Father ) {
     // first unbound handler

    } catch ( fred.Child ) {
     //Even though Father and Child are related,
     // there is no splitting since Father was unbound

    } catch ( mary.Aunt ) {
```

```
    //Again, no splitting since Aunt is not related to Child
    // catch( Aunt ) and catch( Child ) cannot catch the same exception

  } catch ( Child ) {
    //Here, splitting is necessary even though the catch is unbound
    //Reason: Child is related to Child, which was bound to fred before

  } catch ( mary.Father ) {
    //No splitting here. Father may be related to Child, but Child was
    // unbound in the first catch clause of current try-block

  } catch ( fred.Father ) {
    //Theoretically, splitting is necessary
    // (Father and Father are related, and mary.Father bound),
    // but since exception type and declarator (none in original source,
    // U_ BOUND_ PARM after transformation) are equal, optimization is possible
    // -> no splitting is required

  } catch ( john.Father e ) {
    //Here, splitting is required since the declarator e makes
    // optimization impossible

  } catch ( Grandmother e ) {
    //Again, splitting is required since Father and Grandmother are related
    // and john.Father of the same (new) try-block is bound

  }

}//main
```

B.2 Example after transformation

```
// first part of file omitted

...
class Child : public Father {
  private : protected :
    virtual void * uEvent_id ( ) const { return & uId ; }
    virtual uDualClass * uDuplicate ( ) const { return new Child ( * this ) ; }
    virtual void uStackThrow ( ) const { throw * this ; }
  public : static void * uId ;
    Child ( ) { uId = & Father :: uId ; { } }
  private :
} ;
void * Child :: uId = __null ; ;


class someObject {
  private : protected : public : private : public :
    void doSomething ( ) {
     { uOrigRethrow = true ; uAEHM :: uAEThrow ( * Grandmother ( ) .uSetOriginalThrower( this ) ) ; }
    }
} ;

void uMain :: main ( ) {
   uTaskMain uTaskMainInstance ( * this ) ;
   {//1
    someObject fred , mary , john ;
    {//2
        bool uOrigRethrow = false ;
        try {//3
         try {//4
            try {//5
             try {//6
                {//7
                 {//8
                    bool uOrigRethrow = false ;
                    try {//9
                     {//10
                        // after transformation, notice the second variable
                        // uOrigRethrow in this scope overlapping the first one

                        mary . doSomething ( ) ;

                     }//10
                    } catch ( Aunt ) {
                     if ( uOrigRethrow ) throw ; {
```

```
                // unbound Handler, note: uOrigRethrow is checked everywhere
                // after transformation
            }
           }//9
       }//8
      }//7
  }  catch ( Father ) {
      if ( uOrigRethrow ) throw ; {

       // first unbound handler
      }
  } catch ( Child U_BOUND_PARM ) {
      if ( uOrigRethrow ) throw;
      void * _U_bindingVal = ( U_BOUND_PARM ) . uGetOriginalThrower();
      if ( ( _U_bindingVal >= &( fred ) )
        && ( (int) _U_bindingVal < (int) &(   fred   ) + sizeof( fred ) ) ) {

       // Even though Father and Child are related,
       // there is no splitting since Father was unbound

      } else { throw ; }
  } catch ( Aunt U_BOUND_PARM ) {
      if ( uOrigRethrow ) throw;
      void * _U_bindingVal = ( U_BOUND_PARM ) . uGetOriginalThrower();
      if ( ( _U_bindingVal >= &( mary ) )
        && ( (int) _U_bindingVal < (int) &(   mary   ) + sizeof( mary ) ) ) {

       // Again , no splitting since Aunt is not related to Child
       // catch ( Aunt ) and catch ( Child ) cannot catch the same exception

      } else { throw ; }
  }//6
 } catch ( Child ) {
  if ( uOrigRethrow ) throw ;
  {
      // Here, splitting is necessary even though the catch is unbound
      // Reason: Child is related to Child, which was bound to fred before

  }
 } catch ( Father U_BOUND_PARM ) {
  if ( uOrigRethrow ) throw;
  void * _U_bindingVal = ( U_BOUND_PARM ) . uGetOriginalThrower();
  if ( ( _U_bindingVal >= &( mary ) )
       && ( (int) _U_bindingVal < (int) &(   mary   ) + sizeof( mary ) ) ) {

      // No splitting here. Father may be related to Child, but Child was
      // unbound in the first catch clause of current try - block

  } else   if ( ( _U_bindingVal >= &( fred ) )
            && ( (int) _U_bindingVal < (int) &(   fred   ) + sizeof( fred ) ) ) {
```

```
                    // Theoretically , splitting is necessary
                    // (Father and Father are related, and mary.Father bound),
                    // but since exception type and declarator (none in original source,
                    // U_ BOUND_ PARM after transformation) are equal, optimization is possible
                    // -> no splitting is required

                  } else { throw ; }
                 }//5
               } catch ( Father e ) {
                  if ( uOrigRethrow ) throw;
                  void * _U_bindingVal = ( e ) . uGetOriginalThrower();
                  if ( ( _U_bindingVal >= &( john ) )
                    && ( (int) _U_bindingVal < (int) &(  john  ) + sizeof( john ) ) ) {

                   // Here, splitting is required since the declarator e makes
                   // optimization impossible

                  } else { throw ; }
                }//4
              } catch ( Grandmother e ) {
                if ( uOrigRethrow ) throw ; {

                  // Again, splitting is required since Father and Grandmother are related
                  // and john.Father of the same (new) try-block is bound

                }

              }//3
            }//2
          }//1
        }//main
```

Bibliography

[1] Peter A. Buhr. Are safe concurrency libraries possible? *Communications of the ACM*, 38(2):117–120, February 1995.

[2] Peter A. Buhr, Ashif Harji, and W. Y. Russell Mok. *Advances in COMPUTERS*, chapter Exception Handling. Academic Press, 2002.

[3] 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.

[4] Peter A. Buhr and Richard A. Stroobosscher. μC++ annotated reference manual, version 4.9. Technical report, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, August 2001. ftp://plg.uwaterloo.ca/pub/-uSystem/uC++.ps.gz.

[5] Qian Cui and John Gannon. Data-oriented exception handling. *IEEE Transactions on Software Engineering*, 18(5):393–401, May 1992.

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

[7] Intermetrics, Inc. *Ada Reference Manual*, international standard ISO/IEC 8652:1995(E) with COR.1:2000 edition, 1995. Language and Standards Libraries.

[8] ISO/IEC 14882:1998 (E), www.iso.org. *International Standard – Programming Language – C++*, 1998.

[9] ISO/IEC 9899:1999 (E), www.iso.org. *International Standard – Programming Languages – C*, 1999.

[10] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Software Series. Prentice Hall, second edition, 1988.

[11] 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/uSystem/MokThesis.ps.gz.

[12] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.

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

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

[15] UNIX International Programming Languages SIG, Waterview Corporate Center, 20 Waterview Boulevard, Parsippany, NJ 07054. *DWARF Debugging Information Format*, 1993.