TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Matti Rintala

Techniques for Implementing Concurrent Exceptions in C++

Julkaisu 1075 • Publication 1075

Tampere 2012

Tampereen teknillinen yliopisto. Julkaisu 1075
Tampere University of Technology. Publication 1075

Matti Rintala

Techniques for Implementing Concurrent Exceptions in C++

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 2nd of November 2012, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2012

ISBN 978-952-15-2915-3 (printed)
ISBN 978-952-15-2922-1 (PDF)
ISSN 1459-2045

Abstract

In recent years, concurrent programming has become more and more important. Multi-core processors and distributed programming allow the use of real-world parallelism for increased computing power. Graphical user interfaces in modern applications benefit from concurrency which allows them to stay responsive in all situations. Concurrency support has been added to many programming languages, libraries and frameworks.

While exceptions are widely used in sequential programming, many concurrent programming languages and libraries provide little or no support for concurrent exception handling. This is also true for the C++ programming language, which is widely used in the industry for system programming, mobile and embedded applications, as well as high-performance computing, server and traditional desktop applications. The 2003 version of the C++ standard provides no support for concurrency, and the new C++11 standard only supports thread-based concurrency in a shared address space.

Procedure and method calls across address space boundaries require support for serialisation. Such C++ libraries exist for serialisation of parameters and return values, but serialisation of exceptions is more complicated. Types of passed exceptions are not known at compile-time, and the exceptions may be thrown by third-party code.

Concurrency also complicates exception handling itself. It makes it possible for several exceptions to be thrown concurrently and end up in the same process. This scenario is not supported in most current programming languages, especially C++.

This thesis analyses problems in concurrent exception handling and presents mechanisms for solving them. The solution includes automatic serialisation of C++ exceptions for RPC, and exception reduction, future groups and compound ex-

ceptions for concurrent exception handling. The usability and performance of the mechanisms are measured and discussed using a use case application.

Mechanisms for concurrent exception handling are provided using a library approach (i.e., without extending the language itself). Template metaprogramming is used in the solutions to automate mechanisms as much as possible. Solutions to the problems given in this thesis can be used in other programming languages as well.

Preface

Completion of this thesis has been a slow process. It would not have been possible without support and encouragement from many people, who I want to thank wholeheartedly for occasionally kicking my rear end.

I want to thank late Prof. Ilkka Haikala for helping me during my early postgraduate phase and arranging my research sabbatical in London, which gave birth to many themes found in this thesis. I want to thank Prof. Russel Winder from King's College London for his support, ideas, and suggestions during my sabbatical.

Especially I want to thank my supervisor Prof. Kai Koskimies for his continuing support, encouragement and wisdom during the time it took to complete the thesis. I'm also grateful for the opponent Assoc. Prof. Zoltán Porkoláb and thesis reviewers Prof. Heikki Saikkonen and Dr. James Coplien for their valuable comments.

Many thanks go also to all my colleagues at the Tampere University of Technology, who have always been ready to listen, comment, and give advice. Especially I want to thank Jyke Jokinen and Timo Aho for reading and commenting on drafts of this thesis and papers.

Finally the most important. I want to thank, hug, and kiss my wife Jenni for her patience, love, and understanding during my dark hours when I had abandoned all hope. Similar hugs and kisses go to our son Viljami, who has given me more happiness than I can put into words. His birth was the best delay to this thesis I could have hoped for.

Tampere, 6th September 2012

Contents

Figures

Tables

Code listings

Chapter 1

Introduction

The subject of this thesis is mechanisms for concurrent exception handling and implementation of those mechanisms using the C++ programming language.

The thesis studies how to combine concurrency and exception handling, using the C++ language as an implementation platform. This chapter briefly discusses concurrency, exception handling and implementation of these in C++ so that the contributions can be more easily understood. Chapter 2 discusses these topics in more detail. This chapter also lists the main contributions of this thesis and presents its structure.

1.1 Concurrency and asynchrony in programming languages

In recent years, concurrent programming has become more and more important. It is increasingly difficult to produce processors with higher clock frequencies, so increases in computing power are gained by adding more processors or processor cores operating in parallel. Distributed systems and cloud computing are based on several concurrently operating computers. Modern end-user applications use graphical user interfaces which have to stay responsive even if heavy calculations are carried out at the same time. For these and many other reasons most mainstream programming languages have concurrency support of some sort.

The reasons behind the need for concurrency affect the way concurrency is usually introduced into a programming language. This can clearly be seen from literature on concurrent and parallel languages [Wilson and Lu, 1996]. If high performance parallel processing is the goal, then it is important to concentrate on efficient method dispatching, optimised communication between processes and processors, load balancing, etc. Since a lot of high-performance computing uses vector or matrix calculations of some sort, emphasis is often on parallel execution of some operation on a set of data elements. High-performance computations typically benefit from quite low-level and fine-grained parallelism since computations consist of a series of operations performed on a set of data in parallel.

However, if the need for concurrency comes from reactive systems, the situation is quite different. Efficiency is usually not a prime concern, and all concurrently running parts of the program may run on the same processor, making it unlikely that actual performance of the program could be improved by concurrency. However, in these systems concurrency can greatly help in making the program more responsive to outside events. The goal in this case is to design a language environment where the programmer can easily divide the program into several concurrently running parts, and make these parts communicate with each other as naturally as possible. This thesis concentrates mainly on these aspects of concurrency although the performance of solutions is also measured and analysed.

If concurrency is achieved using processes communicating with each other and with no shared memory, any transferred objects or data structures must be embedded in the message and the objects or data must be re-created on the receiving end. This is called *serialisation* or *marshalling*/*unmarshalling*. Some languages like Java provide serialisation as a language feature [Sun Microsystems, Inc., 2006, `java.io.Serializable`], but user-defined classes and data structures still may need their own custom written serialisation routines. Some of the object data can be outside the objects, but that data is still part of the objects' logical state and must be included in serialisation. If a programming language does not offer any support for serialisation, concurrency libraries must provide their own serialisation support from scratch. And if exceptions are to be passed between processes with no shared memory, they have to be serialised as well.

One way to achieve asynchrony is to execute method calls or ordinary function calls in a different thread than the calling thread, and allow the calling thread to continue its execution while the method or function is being executed. If the method

or function has no return value, this scheme can be used transparently in place of an ordinary method or function call. However, return values are usually available only after the call has been completed. The caller has already continued its execution, so the return value cannot be returned as the "value" of the call expression, as usual.

Futures are a mechanism for asynchronous return value passing. They are placeholders for the return value in the caller. Futures were originally introduced in Multilisp [Halstead, 1985], but recently support for futures have been added to many languages, including Java and C++11. When an asynchronous method or function is called, it immediately returns a future object. When the actual return value is available at the end of the asynchronous call, it is transferred to this future. The caller can poll the future to see if a return value is already available, or the caller can wait for the value to become available. This enables the caller to proceed without waiting for a call to complete, but still gives it access to the return value later when it becomes available. If exceptions are used in a concurrent environment where futures are used, futures are affected since an exception replaces the normal return value.

1.2 Exception handling

Exception handling is a mechanism for handling program errors and abnormal or exceptional situations in a structured way. The foundations of modern exception handling were laid in the 1970s [Goodenough, 1975, Ryder and Soffa, 2003]. Exceptional and error situations vary from program to program and domain to domain, so an exact definition of an "exceptional situation" is difficult. In [Buhr and Mok, 2000] the authors state that "Substantial research has been done on exceptions but there is hardly any agreement on what an exception is. Attempts have been made to define exceptions in terms of errors but an error itself is also ill-defined."

In modern programming languages, exceptions and exception handling have largely replaced other methods of signalling about exceptional conditions. However, combining exception handling and concurrency is not trivial, and this thesis analyses and tries to solve problems in concurrent exception handling.

When a program detects an exceptional situation, it raises (throws) an exception. Program execution is then automatically transferred to an appropriate exception handler, which should know how to handle the situation. The exception handler is chosen depending on the type of the raised exception. Each exception handler has its own scope where it is "active"—the handler is considered for receiving the

exception only if program execution is inside the scope of the handler when an exception is raised.

Exceptions help in separating the “normal” control flow from the “exceptional” control flow. Exception handling code is isolated in exception handlers, and the rest of the code contains the normal execution of the program. With exceptions, return values can be reserved to return the results of successful execution, and exception objects are used to return information about an unsuccessful execution.

In object-oriented programming languages exceptions are typically objects, making it possible to embed data about the exceptional situation inside the exception object. This way information can be passed from the raising code to the exception handler. One of the first languages to use this scheme was ML [Milner *et al.*, 1997].

In object-oriented languages inheritance is used to create hierarchies of classes. These class hierarchies can be used in exception handling, which allows exception handlers to accept a group of exceptions polymorphically using base classes in the hierarchy. This combination of inheritance and exceptions was introduced in C++ [Stroustrup, 1993], and is now widely used in other object-oriented languages like Java.

New exception types can be created by deriving new exception classes through inheritance. This is important in this thesis, since it means that exceptions must always be treated polymorphically and new exceptions classes can be introduced in third party libraries etc, which are beyond the control of the programmer.

1.3 The C++ programming language

The C++ language is widely used in the industry. Its ability to produce efficient programs makes it suitable for system programming, mobile and embedded applications, as well as high-performance computing, server and traditional desktop applications. In many areas like embedded systems C++ is replacing the C language because it supports object-oriented programming. On the other hand, at the same time other object-oriented high-level languages like Java are replacing C++ in applications where computing power, memory, and performance issues are not a problem.

The current 2003 version of the C++ language [ISO/IEC, 2003] does not provide any support for concurrency. Concurrency is not mentioned at all in the language standard, meaning that possible concurrency support depends on the used compiler and/or operating system services. The new C++11 standard [ISO/IEC, 2012] provides

basic support for concurrency, but information on that was not available when the work for this thesis was done. Chapters 7 and 8 contain discussion about C++11 in more detail.

Exceptions were added to C++ before its standardisation in 1998, and their use in programs has increased as compiler support for exceptions has improved. Exception handling in C++ works as explained in Section 1.2. However, although exceptions in C++ can form hierarchies through inheritance, this is not mandatory. An object of any class (or even values of primitive types) can be raised (C++ uses the term "thrown") as exceptions. This means that exceptions in C++ do not form a single inheritance hierarchy with a single common ancestor.

The C++ language has recently gained popularity in generic and generative programming and metaprogramming. The C++ template mechanism is a Turing-complete metaprogramming platform which allows the creation of compile-time metaprograms which can (with restrictions) make decisions by inspecting existing types and generate new code based on those decisions [Abrahams and Gurtovoy, 2004]. All this happens during compilation with no run-time penalties.

These metaprogramming abilities make C++ an interesting implementation platform, since template libraries can be used to create code which on many other languages would require extending the language syntax. Many of the mechanisms in this thesis use template metaprogramming.

1.4 Problem statement

While exceptions are widely used in sequential programming, many concurrent programming languages and libraries provide little or no support for concurrent exception handling. This may be because combining concurrency and exception handling is not trivial and has several problems. Most of these problems are caused by the fact that both exception handling and concurrency affect the control flow of the program, sometimes in ways that are hard to combine.

The C++ language has static typing. Return values (as well as parameters) are passed by value instead of by reference. As mentioned before, C++ also has no support for serialisation, so inter-procedural return values have to use custom libraries for marshalling and unmarshalling. In exceptional situations, C++ methods and

functions may throw an exception of any type.[1] This means that despite static typing, marshalling and unmarshalling exceptions require dynamic and polymorphic creation of exception objects on the receiver. Several serialisation libraries exist for C++ [OMG, 2003, Ramey, 2004, Free Software Foundation, Inc., 2004, s11n, 2005], but their support for serialised exceptions is limited.

In addition to this, an exception thrown from a method or function may be originally thrown deeper in the program, possibly by a third-party library. This means that not only can the exception be of any type, it may also be of a type the programmer has no control over and whose code the programmer cannot modify. Passing these exceptions to another process through serialisation should also be possible.

All this makes serialisation of exceptions more problematic than serialisation of parameters or return values. Serialisation libraries use explicit library calls to perform serialisation, but this requires both knowledge on the type of the exception (so that it can be passed to the library) and catching the exception so that it can be passed to the serialisation code. Serialisation in turn requires type-specific marshalling routines which have to be called polymorphically since the exact type of the exception is not necessarily known.

Asynchrony adds additional problems to concurrent exception handling, and current programming languages differ in their approach to concurrent exception handling [Romanovsky and Kienzle, 2001]. Traditional sequential exception handling ties together the control flow and handling of exceptional conditions because exceptions automatically transfer execution to an exception handler. Which exception handlers are enabled in each case depends on where the thread of control in the program was when the exception was thrown. In a concurrent program, the caller may already have left the scope of relevant exception handlers when an exception is received from an asynchronous call. This makes it unclear which exception handlers should be considered for exception handling.

Concurrency also introduces the possibility of several concurrent exceptions, if the caller is already in the middle of exception handling when another exception is received from an asynchronous call. Similarly several exceptions can be received from several asynchronous calls concurrently. These situations do not normally occur in sequential programs, or they are explicitly forbidden (for C++ this is further

[1] In theory C++ has *exception specifications* which can be used to restrict allowed exceptions to certain types. However, this is checked only during run-time, and it has problems with generic exception-neutral programming. For these reasons, exception specifications are declared a deprecated feature in the new C++11. [ISO/IEC, 2012, §D.4]

discussed in Section 2.2.3). In distributed systems there has been research on this topic [Xu *et al.*, 2000].

One possibility for concurrent exception propagation is to tie return values and exceptions together, i.e. an asynchronous call returns either a value or an exception. If an exception is returned, it is activated when the returned "value" is accessed. This approach still contains both the problems discussed earlier, but it adds some new problems. If thrown exceptions are returned as values, they can also be copied and passed around. This creates the possibility to duplicate exceptions or to activate their handling in a different place and at a different time than would otherwise happen.

The topic of this thesis is concurrent exception handling mechanisms, with emphasis on the C++ language. The aim of the thesis is to provide solutions to concurrent exception handling problems using a library approach (i.e., without extending the language itself). The aim is also to provide solutions that can be used directly with third-party libraries without having to make them concurrency aware. The solutions should not depend on any specific concurrency library or model (like active objects) although reference implementation and case studies are implemented in author's KC++ system.

Solutions to the problems given in this thesis can be used in other programming languages, but applicability of the solution mechanisms depends on the concurrency and object model of the language, as well as its metaprogramming support. Applicability of each mechanism to other programming languages is discussed in each relevant chapter.

Figure 1.1 on the following page gives a summary of concepts that are central to the problems discussed in this thesis, and it also shows how the concepts relate to each other. The starting points "concurrency" and "exception" are marked in yellow, intermediate concepts in orange, and main problem concepts in red and purple. Arrowheads show the direction in which connections are meant to be read.

1.5 Thesis approach

This thesis describes how exception handling and concurrency can be combined in the C++ programming language. C++ was chosen because of it is used in high-performance computing as well as low-power devices, which both benefit from con-

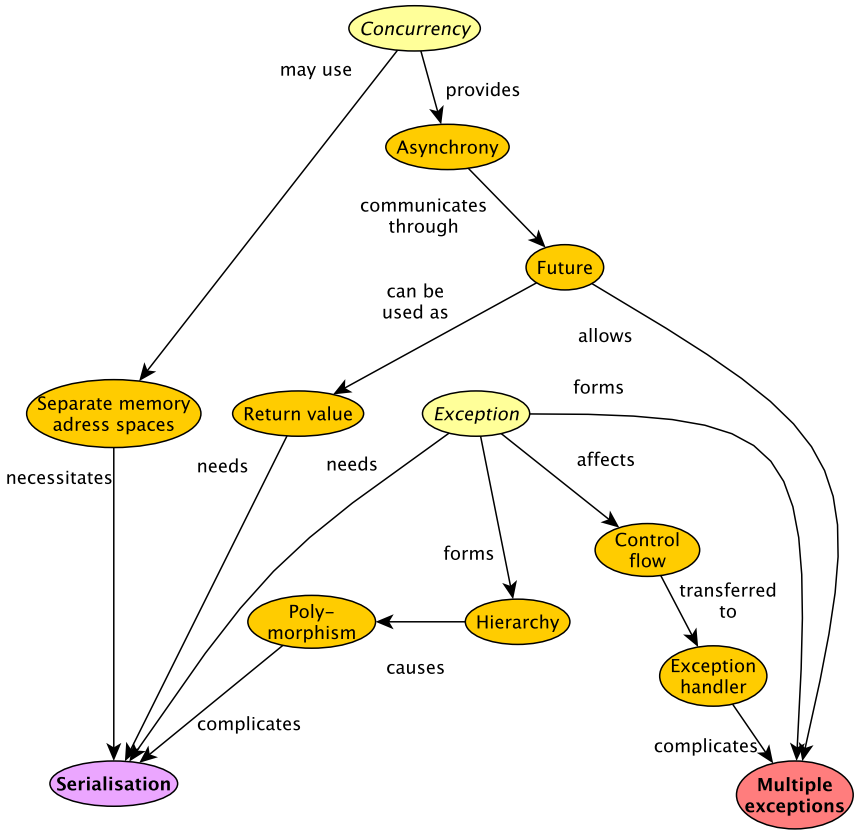

FIGURE 1.1: *Diagram showing thesis problem concepts and their connections*

currency and parallelism. The metaprogramming facilities of C++ provide opportunities for implementing many of the introduced mechanisms as a library instead of writing a custom compiler and extending the language.

Before the C++11 standard the language did not contain any support for concurrency. The new standard did not exist when the main work for this thesis was done, so this thesis uses the concurrent object oriented KC++ system as its implementation platform to provide basic concurrency.

KC++ is a concurrent library and preprocessor for C++ written by the author. The C++ syntax is not modified, but some additions are made to the semantics to allow con-

currency. KC++ was introduced in the author's licentiate thesis [Rintala, 2000]. The history of the KC++ system began in 1998 during the author's stay in the King's College London under the supervision of Prof. Russel Winder, who had earlier been working on UC++, another C++-based concurrent programming system [Winder *et al.*, 1996]. The goal was to continue on the path of UC++ but design a new system which would better integrate into the C++ language. The concurrent exception handling described in this thesis was developed while working on the KC++ project, so KC++ was chosen as an implementation platform for the mechanisms.

However, the choice to use KC++ was made only because it allowed case studies and code examples to be concrete so that they can be compiled, tested and evaluated. The mechanisms presented in this thesis do not depend on the KC++ system, nor its model of concurrency (active objects). Each main chapter contains discussion on programming language features the mechanisms depend on and require.

The approach used in this thesis is constructive. The problem area of concurrent exception handling is approached by identifying problems and limitations in sequential exception handling. Then solutions to those problems are designed and discussed, and those solutions are implemented in the platform used in this thesis, KC++. Finally real-world applicability and performance of those solutions are tested by designing and writing a case study application. The case study includes discussion on the suitability of the mechanisms presented in this thesis as well as tests to measure the performance impact the mechanisms have on exception handling in the case study.

KC++ embeds concurrency features into the C++ language as naturally as possible, so that programmers can utilise them using conventional programming techniques and idioms. The goal has also been to make sure that concurrency features do not needlessly restrict the use of other language features.

Concurrency in KC++ is achieved through futures and active objects which are described in Chapter 2. *Active objects* are one way to express concurrency in an object-oriented environment [Lavender and Schmidt, 1995]. An active object is an object with its own thread of execution. This is used to execute methods of the object. Other threads calling a method on the object just request that method to be executed in the thread of the active object. In some approaches, an active object may create a new concurrent thread for each method call, allowing concurrency inside the active object.

The mechanisms described in this thesis have been implemented as a conventional C++ library rather than as a language extension. This makes it easier to implement the mechanisms in other concurrent C++ environments. It is also in line with the design principles of C++, where library components are preferred instead of core language extensions [Stroustrup, 2007, §8.1]. Template metaprogramming and other C++ techniques are used to compensate for the lack of run-time reflection capabilities in C++ and to make using KC++ as straightforward as possible.

Although KC++ is targeted towards areas where program speed is not the most important criteria, it is still important that the mechanisms described in this thesis can be implemented efficiently enough, so that the higher-level concurrency constructs do not add an unacceptable overhead to program execution. For this purpose a working prototype of the KC++ system has been constructed and measurements have been taken to estimate the overhead caused by concurrent exception handling.

It should be emphasised that high-level concurrency features are often slower than low-level small-grain mechanisms. Parallel performance of a system also depends much on how the system internally handles message passing, load-balancing and other mechanisms which are hidden from the programmer using the system. Therefore it was not regarded reasonable to experiment with performance tests against other concurrent C++-based systems.

To get a perspective of concurrent exception handling in practise, this thesis includes a concurrent exception-aware implementation of the Observer design pattern [Gamma *et al.*, 1996, Ch. 5] as a case study. This implementation is used to discuss problems relating to concurrent exception handling in applications.

Even though the mechanisms in this thesis are currently implemented on the KC++ platform, they are not restricted to that platform. As the mechanisms are library based, they could be implemented on many other concurrent C++ systems as well. The ideas and mechanisms could also be used in other languages, but that depends on the concurrency and object model and metaprogramming support provided by the language.

Figure 1.2 on the next page shows a concept map expanded from Figure 1.1, describing concepts related to the contributions and their connections to the problem concepts. Colours and connections are as in Figure 1.1.

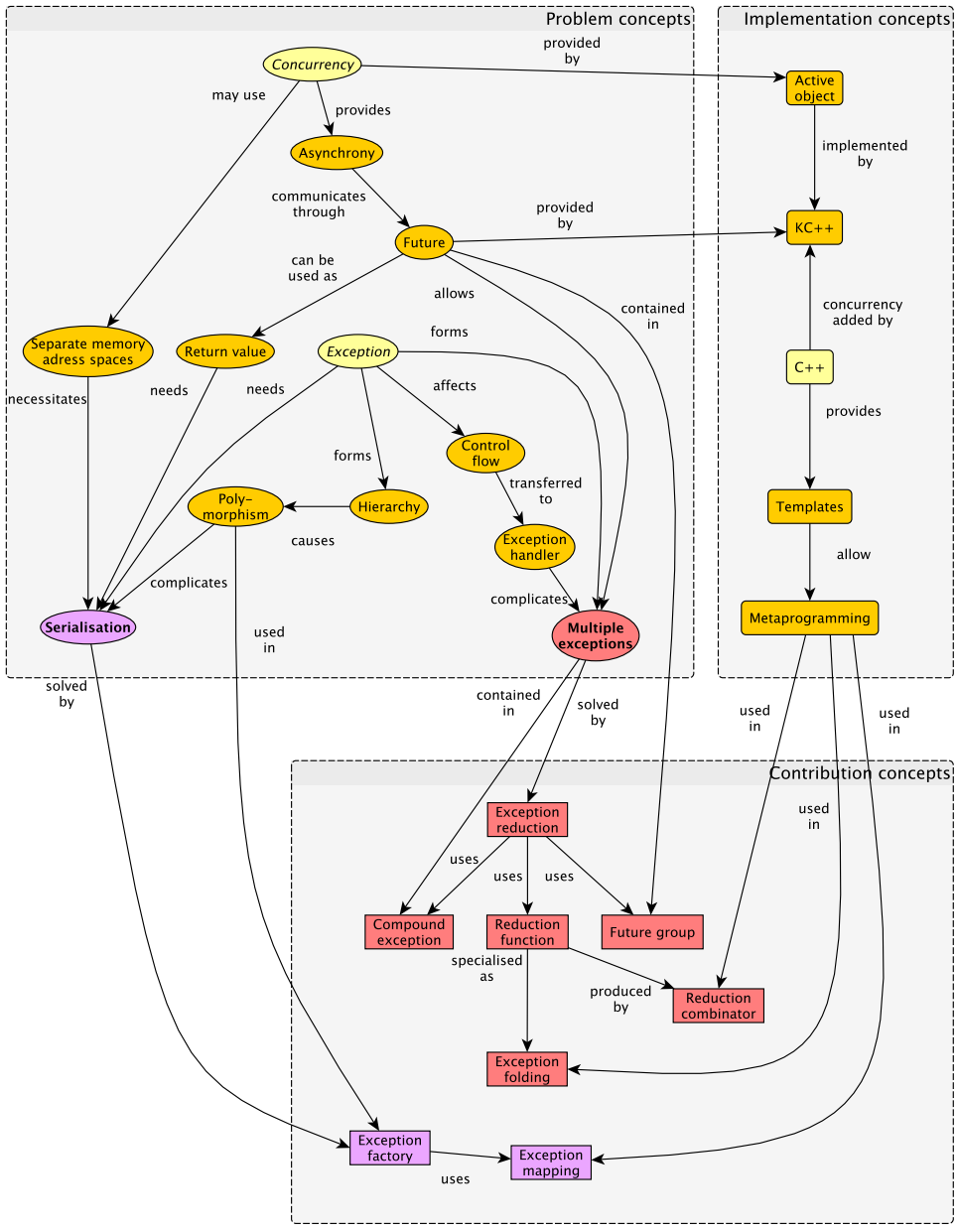

FIGURE 1.2: *Concepts related to the thesis, with connections*

1.6 Contributions

The main contributions of the thesis are as follows:

- **Automated serialisation of RPC exceptions.** This thesis shows how exceptions can be propagated among processes with no shared memory. This includes dynamic creation of exception objects on the receiving end using an exception factory as well as mapping of non-RPC capable exceptions to their RPC capable counterparts. This is automated using template metaprogramming and can be done without modifying existing exception classes, making it usable with third party libraries. The approach can be implemented as a library with no language extensions. Parts of these contributions have been published in [Rintala, 2007].

- **Handling and reduction of multiple concurrent exceptions.** Mechanisms are provided for handling concurrently raised exceptions. These include compound exceptions for grouping such exceptions together, future groups for synchronising with such exceptions, and reduction functions for analysing and reducing a group of exceptions. These mechanisms are originally published in [Rintala, 2006]. In addition to this, this thesis also discusses metaprogramming based combinators which allow building reduction functions from ready-made parts, and reduction by folding exceptions together based on inheritance hierarchies.

- **A case study and performance analysis.** The concurrent exception handling mechanisms are used to write a concurrent implementation of the Observer design pattern, which in turn is used to implement a case study application. This case study is used to discuss and evaluate concurrent exception handling using mechanisms presented in this thesis. Performance of the mechanisms is then measured and analysed using the application. Also the low-level performance of the RPC exception passing mechanism is measured and discussed.

1.7 Structure of the thesis

The structure of this thesis is the following:

- Chapter 2 contains necessary background information. It gives an overview on concurrency, futures, C++ exception handling, and C++ template metaprogramming.
- Chapter 3 covers asynchrony, futures, and active objects in the KC++ system that was used as a platform for the main contributions of this thesis. This chapter contains information needed to understand the platform on which the case studies are built.
- Chapter 4 addresses problems related to propagating C++ exceptions among processes running in separate address spaces. This chapter also introduces the structure of RPC exception hierarchies and mapping of non-RPC exceptions to their RPC counterparts. This chapter is based on a published article "Exceptions in remote procedure calls using C++ template metaprogramming" [Rintala, 2007].
- Chapter 5 describes how concurrent exception handling can be implemented, including the implications of multiple concurrent exceptions. The chapter also describes mechanisms to help the programmer with concurrent exception handling and exception reduction. This chapter is based on a paper "Handling Multiple Concurrent Exceptions in C++ Using Futures" [Rintala, 2006].
- Chapter 6 contains a case study implemented using the KC++ system and its evaluation. The chapter also provides measurements for determining how the mechanisms of this thesis affect program performance.
- Chapter 7 presents relevant related work by others. It also discusses how new features of the new C++11 standard affect the contributions of this thesis.
- Chapter 8 concludes this thesis by summing up the experiences learned from KC++. It also discusses some future directions for further development of the ideas described in this thesis.

A proof-of-concept implementation exists for mechanisms and the case study presented in this thesis. Relevant parts of them are included in the thesis as code

listings, but because of size constraints all source code is not included. A hyperlinked browsable version of the source code is available on the net, and links to it are provided in relevant chapters.

Chapter 2

Background information

This chapter contains background information necessary to understand contributions of this thesis. Basic concepts of concurrency, asynchrony, active objects, exception handling, and template metaprogramming are presented (template metaprogramming is used in the mechanisms of this thesis).

The reader is expected to have basic knowledge on exception handling, and some experience with the C++ language. Deeper knowledge of concurrency, exception handling, metaprogramming, or details of C++ implementation is not required, necessary information on these topics is given in this chapter.

2.1 Concurrency

Concurrency has been an important topic in computer science for decades, but its importance has grown even more in the past few years. Multicore CPUs are now mainstream even in laptops and home workstations, and applications like photo and video processing require and increasing amount of processing power.

It is common practise to use the word “parallelism” to refer to several processors executing a program at the same time. The word “concurrency” in turn means that several tasks are in execution at the same time, even if only one processor is present and only one task proceeds at any time [Andrews, 1991]. Thus parallelism always includes concurrency but concurrency is possible without parallelism. Some sources however use words “parallelism” and “concurrency” interchangeably

[Magee and Kramer, 1999] (sometimes terms “true parallelism” or “hardware parallelism” is then used to refer to parallelism using several physical processors).

A concurrent program consists of several concurrently executing threads of execution. In order to benefit from concurrency, these threads often execute most of the time independently, in asynchronous fashion. A program may also contain synchronisation points where two or more concurrently executing threads wait for each other. One common purpose for synchronisation is to transfer information from one thread of execution to another.

Concurrent threads of execution may share a common address space and use common data structures, in which case they are usually referred to as *threads*. Alternatively, they can each execute in their own address space, in which case they are commonly called *processes*. In the first case information can be exchanged between threads using shared data structures. In the second case processes have to use alternative methods like message passing to communicate with each other.

If concurrent processes do not have shared memory, any objects and data passed between the processes must be transmitted using message passing or some other mechanism. The sender *marshals* the object into a series of bytes. Those bytes are transmitted to another process, which then *unmarshals* those bytes, creating a copy of the original object. Since this serialisation must include all data logically belonging to the object, the serialisation of user-defined objects requires that programmer somehow indicates which data must be included in serialisation.

In the simplest form this can be achieved with user-defined marshalling and unmarshalling functions or methods. These perform the mapping from an object to a data stream and back. From an object-oriented viewpoint, marshalling methods belonging to the object to be serialised would be logical. However, if marshalling has to be added to a third-party class, separate marshalling functions allows doing this without modifying the class itself.

When concurrency is introduced into object-oriented programming, it is quite natural to attempt to merge it with objects — after all, object-oriented programming *is* about objects. This leads to the concept of *active objects*. Active objects are objects whose methods are executed concurrently with the rest of the program. The idea of active objects is common enough to be classified as a design pattern [Lavender and Schmidt, 1995].

Object level is not suitable for very low-level concurrency. For example, it is not reasonable to model individual machine code instructions etc. as “objects” and

try to execute them in parallel. However, object level is quite usable in high-level concurrency. Encapsulation isolated objects from the rest of the program, and this encapsulation makes it possible to put the internals of an object in its own address space — on another processor with its own memory or even a remote machine. At the same time, encapsulation enforces the idea of strict interfaces as the only way to use an object. These interfaces provide a natural way of communicating between two concurrently running parts of a program. This kind of encapsulation resembles Hoare's monitors [Hoare, 1974]. It should be noted, however, that the active object approach is not the only possibility, but concurrency can be introduced as a mechanism completely independent of objects.

In a concurrent environment, especially with active objects, asynchrony is inherent. When a method of an active object is called, the caller continues its execution, while the active object begins executing the method. If later the caller needs to know that the method has been completed, some sort of synchronisation mechanism is needed. Several such mechanisms exist, ranging from low-level semaphores [Dijkstra, 2002] and barriers [Jordan, 1978] to condition variables inside monitors [Hoare, 1974], rendez-vous in Ada [Ada, 1995, §9.5.2(25)], and other higher level constructs.

In addition to knowing that a method has been completed, there is often information to be passed back from the method to the caller. This information can be a traditional return value or a two-way "in-out" parameter (a reference parameter in C++). *Futures* [Halstead, 1985] are one way for the caller to have placeholders for values that will become available later when the method has been completed. Futures can be copied, assigned to each other and passed around without having to wait for their eventual value. When the value becomes available, it is automatically propagated everywhere the future was copied.

In this thesis, the KC++ language [Rintala, 2000] uses futures as value passing mechanism. Futures can also be used as a synchronisation mechanism in methods with no return value.

It should be noted that asynchrony can be achieved even without concurrency. For example, in *lazy evaluation* expression are evaluated and functions are executed only when and if their value is needed. This behaviour is similar to non-hardware concurrency in that function execution does not happen immediately when it is called, but is executed later (in fact, futures in the C++11 standard can be used both for concurrent and lazy execution). Exception handling causes problems in lazy evaluation also, but they are out of the scope of this thesis.

2.2 Exception handling

Programmers have always had to prepare for exceptional situations like invalid inputs, timeouts, I/O errors, running out of memory or even error situations caused by bugs in the program. In early programming languages the program logic for reporting and responding to these situations was written using special return values, error codes, global error flags, etc. However, this was error prone and impaired readability of programs. For these reasons programming languages started to provide structures for handling exceptional situations.

Exception handling is a mechanism for handling control flow in exceptional situations and it has become widespread in modern programming languages. One of the first papers to describe and analyse exception handling was written in 1975 by John B. Goodenough [Goodenough, 1975]. Exception handling began in languages like PL/I, Mesa, and CLU, and was later adopted to languages like Ada, Smalltalk, Modula-3, C++, and Java.

2.2.1 Exception handling in C++

Exception handling in C++ resembles exception handling in Ada and Java (which got its exception handling partly from C++). C++ exceptions are objects (or values) which are *thrown* in code where the exceptional situation is detected. An exception in C++ can be of any type, i.e. an object, a primitive type, a pointer, etc. In practise, usually only objects are thrown, and these objects belong to exception classes specifically written to model exceptional situations that may happen in the program. The act of throwing the exception may copy it, because the lifetime of the original object may end before exception handling is complete. The compiler is also allowed to optimise away copying. Listing 2.1 on the facing page shows an example of C++ exception handling, where an object of user-defined class `MyException` is thrown on line 7.

After the exception has been thrown, an appropriate exception handler—a *catch block*—is searched by matching the type of the thrown exception object to the parameters of the catch clauses active at the time. Each catch block in the program is eligible only if program execution has entered and not yet left the *try block* associated with the catch block. If several catch blocks match the thrown exception, the one with most recently entered try block is chosen. Program execution is then transferred to the exception handler in the catch block. Listing 2.1 contains a catch block on lines 9–12 and its try block on lines 5–8.

```
1  class MyException { /* ... */ };
2
3  void f()
4  {
5    try
6    {
           ⋮
7      if (condition) { throw MyException(); }
           ⋮
8    }
9    catch(const MyException& exp)
10   {
11     // Exception handling
12   }
13 }
```

LISTING 2.1: *Example of C++ exception handling*

At the end of the handler, the exception is considered handled, the copy of the exception object is destroyed, and program execution continues from the code after the try-catch compound containing the exception handler. Alternatively, the exception handler may throw another exception or re-throw the current exception. In both these cases exception handling is restarted. In the case of re-thrown exceptions, exception handling continues as if the exception object was originally thrown from the exception handler re-throwing it. In Listing 2.1 the copy of the exception object thrown on line 7 (and referred to by reference `exp` in the catch block) is destroyed on line 12.

When a matching exception handler is searched for, inheritance hierarchies and the polymorphic type of the thrown exception are taken into account. This means that a catch block whose parameter is a base class of the thrown exception is a match. According to Bjarne Stroustrup, C++ took the idea of exception hierarchies from the ML language [Stroustrup, 1994, p. 387]. Exception hierarchies allow programmers to choose the level of abstraction appropriate for each exception handler.

2.2.2 Special features of C++ exception handling

C++ does not require exception classes to form a single inheritance hierarchy (although the C++ standard library provides a small exception hierarchy it uses itself). However, C++ defines a special "ellipsis" catch block **`catch(...)`**, which matches any thrown exception, and can be used to catch all thrown exceptions.

If no handler matches a thrown exception, C++ library function `terminate()` is called and program execution is aborted. A program can register its own termination handler, but even then program execution must be aborted.

When an exception is thrown and program execution is transferred to the appropriate catch block, lifetimes of stack-based local objects may end. This includes local objects declared in the try block associated with the catch block, and any functions and blocks entered from that try block. C++ guarantees that all these local objects are destroyed before the catch block is entered. This is called *stack unwinding*, and it includes executing the destructors of objects that are destroyed.

In many programming languages normal program execution is interrupted when an exception is thrown, and is resumed in the exception handler when the exception is caught. No user code can be executed between these two points. C++ destructors make an exception (sic) to this rule. When an exception is thrown, stack unwinding is performed while the exception is "in transit".[1] This causes limitations on exception handling in destructors (these limitations are discussed in the next section).

C++ standard library provides a way to detect whether stack unwinding is in progress. Function `uncaught_exception()` returns true, if an exception has been thrown but not yet caught [ISO/IEC, 2003, §18.6.4]. In theory this could be used to detect if it is safe to throw an exception from a destructor. However, in practise usefulness of this function is questionable. The reasons for this are also discussed in the next section.

Current C++ compilers are able to implement exception handling so that its impact on program performance is zero when no exceptions occur. This means that no run-time bookkeeping is needed to record the location of try and catch blocks. Only when an exception is thrown, exception handling code analyses return addresses stored on the stack. It uses compile-time lookup tables to find candidates for exception handlers and then tries to find a match for the type of the thrown exception. A detailed explanation of these mechanisms can be found in [ISO/IEC WG21, 2006]. One result of the zero-overhead strategy is that when an exception is thrown, exception handling is a relatively costly operation in C++. This is empirically verified in Chapter 6.

[1] Java also allows executing user code between throwing an exception and entering the catch block. In Java this happens by associating try blocks with "finally" blocks which are always executed even if the try block is exited because of an exception.

2.2.3 Limitations in C++ exception handling

Many of the special features presented in the previous section cause also limitations in C++ exception handling. This section discusses limitations which have impact on the mechanisms introduced in this thesis.

Lack of single exception hierarchy

The C++ language does not force exception classes to form a single inheritance hierarchy. The C++ standard library does provide a class called `exception` and a small hierarchy derived from it, but the language itself does not force programmers to use it. Any object can be thrown regardless of whether it is a part of an inheritance hierarchy or not. In fact, even primitive types like integers or pointers may be thrown although that would normally be considered poor programming style.

The lack of a single exception hierarchy has one severe drawback. It means that there is no general way to polymorphically refer to an arbitrary exception (which would be possible by having a pointer to the top of a single hierarchy). C++ allows the programmer to catch *any* exception using special syntax **`catch(...)`** (where ellipsis `...` is part of the syntax), but such handler has no exception parameter, it has no access to the thrown exception object itself.

The lack of a common base class for exceptions makes it also impossible to create a general container for stored exceptions or to pass any exception as a parameter to a function, etc. All these can be achieved if the program itself provides a single exception hierarchy, but exceptions originating from libraries and third party modules are still a problem.

Copying exceptions

When an exception is thrown, the original exception is automatically copied so that the language run-time can control the lifetime of the exception (copying can be elided by optimisation). This copying is necessary, but it causes some drawbacks and limitations as well.

One limitation is that the programmer has no control over the lifetime of the copied exception object. It is created by the throw statement and destroyed when the catch handler is exited. The catch handler may also re-throw the exception object with syntax **`throw;`**, in which case the lifetime of the exception object continues and the next suitable catch handler is searched for.

If the exception object should be stored for later in the catch handler, the original exception object cannot be used because of its limited lifetime. The only option is to copy the exception object. If the type of the parameter in the handler is the same as the real type of the exception object, copying poses no problems. However, if the catch handler caught the exception using a base class reference, C++ provides no way to copy the object based on its dynamic type. This means that if copying is attempted, a sliced copy may result. (*Slicing* happens when an object is copied using static typing, and the type of the copy is a base class of the original object. The result of slicing is a copy of only base class part of the original object.)

Throwing a stored exception object would pose another problem. When an exception is thrown, automatic copying of the object is also based on static typing. If and when stored exceptions are handled using base class pointers and references, this would again cause slicing.

Exceptions and destructors

When an exception is thrown, stack unwinding leaves scopes and destroys objects whose lifetime ends, executing their destructors. This means that user code may be executed after throwing an exception but before entering a catch block. This ability has some side effects that are discussed in this section.

When a destructor is called during stack unwinding, execution of user code creates a possibility to throw additional exceptions. The C++ standard specifies that this is allowed as long as those exceptions are handled in the destructor, i.e. the destructor exits normally and not via an exception. In essence, this means that nested exception handling is supported, but two exceptions are not allowed to exist on the same level of exception handling.[2] If this rule is violated, library function `terminate()` is called, which in turn terminates program execution [ISO/IEC, 2003, §15.5.1].

In practise this means that that destructors should not end their execution with an exception, either by throwing or by not handling exceptions thrown by functions called in the destructor. This behaviour is required by the C++ standard library and is warned about in many C++ books and style guides [Meyers, 1996, Stroustrup, 2000, Sutter, 2000].

[2] In comparison, Java specifies that if an exception is thrown in a "finally" block, it replaces the original exception which is silently discarded. [Gosling *et al.*, 2005, §14.20.2].

In theory C++ provides a way to detect whether it is safe to throw an exception from a destructor. C++ standard library function `uncaught_exception()` returns true, if an exception has been thrown, but an appropriate catch block has not yet been entered. If it returns false, no exceptions are active.

The problem with `uncaught_exception()` is that it may be safe to throw an exception from a destructor even if the function returns true. Even if stack unwinding is in progress, it is possible that the destructor being executed has its own exception handlers which are able to handle the thrown exception to completion, in which case no exceptions escape the destructor. In other words, when checking whether throwing is possible, `uncaught_exception()` gives a false negative in some cases.

Herb Sutter discusses problems with `uncaught_exception()` in More Exceptional C++ [Sutter, 2001, Item 19]. If a destructor should throw, but cannot do so due to an already thrown exception, it is difficult to find a suitable alternative action. Even if such alternative action was found, it would mean that the program had to handle two errors instead of one. For this reason style guides usually recommend destructors which always return normally without throwing an exception. Use of `uncaught_exception()` is also discussed in [Henney, 2002].

If an exception has been thrown, and a destructor is executed during stack unwinding, C++ provides no way to access the exception object in the destructor. This means that the destructor has no information about the type of the exception, making it even more difficult to decide what to do when throwing another exception is no longer an option.

Exceptions and concurrency

C++ provides no support for serialisation of objects. If exception objects should be copied from one process to another (or even one machine to another), serialisation code and infrastructure must be provided by the programmer. This is done in Chapter 4 in this thesis.

Similarly since current C++ provides no support for concurrency, it provides no support for concurrent exception handling either. All necessary mechanisms must be written from scratch, and even then limitations in the C++ exception handling make concurrent exception handling challenging. This is the topic of Chapter 5.

2.3 C++ template metaprogramming

Templates have made C++ a popular language for generic programming. Templates allow writing generic libraries which work with user-defined types. The C++ standard library is heavily based on templates (especially containers and algorithms in the STL library).

After templates had been introduced to the language, it became apparent that their expressive power was beyond simple type replacement. Template specialisation makes it possible to use templates as a Turing-complete functional compile-time programming language [Veldhuizen, 2003], allowing limited computation and adaptation during compilation. These *template metaprogramming* techniques have been under active study and development during the past few years [Järvi, 2000, Alexandrescu, 2001, Abrahams and Gurtovoy, 2004].

2.3.1 Template instantiation and specialisation

C++ templates are a compile-time facility allowing creation of parametrised classes, structs and functions. Template parameters are either types or compile-time constants. For class and struct templates, the template is *instantiated* by providing the template with appropriate parameters. The result of instantiation is a new type (a class or struct). Function templates are instantiated by calling them, in which case the compiler deduces template parameters from types of actual parameters used in the call (they can also be given explicitly). The result is a function, which is subsequently called.

Template code is not fully compiled until it is instantiated. Full syntactic correctness of a template is checked only during instantiation, since the values of type parameters are not known beforehand. Even before instantiation, the compiler is required to ensure that the template can be parsed properly, and that code not depending on template parameters is syntactically correct.

Class templates use *lazy instantiation*. When a class or struct template is instantiated, its member functions are not instantiated unless they are used. This helps keeping the amount of produced machine code down. It also means that individual members of a class template can have additional requirements on the template parameters, and these requirements are checked only if the member itself is used and instantiated.

Most C++ metaprogramming techniques are based on *template specialisation*. In addition to its definition, each C++ class template may have an arbitrary number of template specialisations, which are used instead of the main template definition for certain values of template parameters. The template for which specialisations are provided is called the *primary template* of those specialisations.

Explicit specialisations provide an alternative definition for the primary template for specific values of template parameters, i.e. a specialisation is chosen instead of the primary template if the provided template parameters match the parameters given in the specialisation. Listing 2.2 shows an example of explicit template specialisation.

Partial specialisation matches a set of template parameters by defining type patterns which the actual template parameters must match. These patterns are formed by defining new template parameters and using these to specify matching parameter values. Listing 2.3 on the following page shows two examples of partial template specialisation.

Lines 1–6 contain the primary template which is used by default. Lines 7–11 provide a partial template which is used if the type parameters are same. This is accomplished by introducing a new type parameter `T` and using pattern `<T, T>` to define that both parameters of the template have to be the same.

Similarly lines 12–17 define a partial specialisation used when the first parameter is **`int`** and the second is a pointer. When the template is instantiated with such parameters, template type deduction is used to find a suitable value for type parameter `T` so that the pattern in the specialisation matches.

```
1  template <typename T> // Primary template
2  struct X
3  {
4    T a;
5  };
6
7  template <>
8  struct X<int> // Explicit specialisation for int
9  {
10   long int a;
11 };
12 X<double> x1; // Instantiates primary template
13 X<int> x2; // Uses explicit specialisation
```

LISTING 2.2: *Example of explicit template specialisation*

```
1  template <typename T1, typename T2> // Primary template
2  struct X
3  {
4    T1 a;
5    T2 b;
6  };
7  template <typename T> // Partial specialisation: both parameters are the same
8  struct X<T, T>
9  {
10   T ab[2];
11 };
12 template <typename T> // Partial specialisation: first parameter is int and
13 struct X<int, T*>        // second parameter is a pointer
14 {
15   long int a;
16   T const * b;
17 };
```

LISTING 2.3: *Example of partial template specialisation*

If template parameters in an instantiation match an explicit specialisation, that specialisation is used instead of the primary template. Similarly, if template parameters in a partial specialisation can be chosen so that the partial specialisation matches the given template parameters, that partial specialisation is chosen. If more than one specialisation matches the given template parameters, the C++ standard provides partial ordering rules for specialisations [ISO/IEC, 2003, §14.5.4.2]. If one specialisation is strictly a better match than others, it is chosen. Otherwise the compiler reports the ambiguity with an error message.

2.3.2 Writing metafunctions using template specialisation

A *metafunction* is a program construct which operates on program code—types, functions, etc. [Czarnecki and Eisenecker, 2000, Ch. 10]. C++ templates allow creation of compile-time metafunctions which take types or compile-time constants as parameters and produce types or compile-time constants as results. This allows automatic generation and selection of types based on other types in the program.

C++ templates can be used to write compile-time metafunctions by writing class templates (or struct templates) whose contents are calculated from the template parameters. Types are "returned" from a template metafunction by member typedefs (a typedef called `type` is often used by convention). Similarly enums with specified numeric values can be used to return compile-time constants. Since each result is

a named type or enum inside the template, several results can be provided by the same template metafunction.

Template specialisation provides a way to implement compile-time selection: during template instantiation an appropriate specialisation (or the primary template) is selected based on template parameters. Each specialisation can produce its result differently, and the primary template acts as a "default" clause if no specialisation matches. Listing 2.4 shows a metafunction that can be used to convert signed types to unsigned and vice versa.

This simple metafunction consists completely of template specialisations. The primary template (lines 2–3) is just a declaration without definition. Then a specialisation is written for each possible value of the type parameter, and these specialisations provide the return values of the metafunction as typedefs. Lines 5–10 define results for **int**, lines 12–17 for **unsigned short int** etc. Finally line 22 shows

```
1  // Empty generic case (compiler error if used)
2  template<typename T>
3  struct SignConvert;
4  // Specialisation for int
5  template<>
6  struct SignConvert<int>
7  {
8    typedef int signed_type;
9    typedef unsigned int unsigned_type;
10 };
11 // Specialisation for unsigned int
12 template<>
13 struct SignConvert<unsigned short int>
14 {
15   typedef short int signed_type;
16   typedef unsigned short int unsigned_type;
17 };
       ⋮
18
19 // Example: Function always returns a signed type
20 // Metafunction is used to calculate return type
21 template<typename T>
22 typename SignConvert<T>::signed_type minus(T a, T b)
23 {
24   return a-b;
25 }
```

LISTING 2.4: *A template metafunction doing compile-time selection*

how the metafunction is called to calculate the return type of a function template.[3] If the template metafunction is instantiated with a type for which no specialisation exists, the primary template is selected and a compiler error is given for a missing definition.

Partial template specialisation makes it possible to write a metafunction which selects from two given types depending on a given condition. Listing 2.5 contains a template `MetaIf` which returns its first type parameter if a compile-time condition is true, otherwise it returns the second type parameter. It also contains an example which shows how template metafunctions can be used in practise.

Partial specialisation uses the same type matching as function templates to deduce its template parameters. Thus it can be used to match only to a certain pattern of types. Similarly, templates can instantiate themselves with different parameters, allowing recursion. This makes templates powerful enough to perform arbitrary compile-time computations.

Listing 2.6 on the next page shows a metafunction which strips any sequence of pointer indirections from a type. In the example, the primary template (lines 2–6) just returns its template parameter as a typedef. Partial specialisation on lines 9–13 is used for pointers. When this partial specialisation is chosen, pattern `Pointee*` is matched to the pointer type used in instantiation. This causes `Pointee` to become the type the pointer type points to. The specialisation then invokes `RemovePointers`

[3] The **typename** keyword in front of the template instantiation is required to tell the compiler that `signed_type` denotes a type. This info is needed to parse the code correctly before the template is instantiated.

```
1  // Generic case for true
2  template<bool condition, typename TrueType, typename FalseType>
3  struct MetaIf
4  {
5    typedef TrueType type;
6  };
7  // Partial specialisation for false
8  template<typename TrueType, typename FalseType>
9  struct MetaIf<false, TrueType, FalseType>
10 {
11   typedef FalseType type;
12 };
13
14 // Example: select int if it is at least 32 bits, otherwise long
15 typedef MetaIf<sizeof(int) >= 4, int, long>::type IntLeast32;
```

LISTING 2.5: *A metafunction selecting from two types*

```
1  // Generic case, return the type parameter directly
2  template <typename T>
3  struct RemovePointers
4  {
5    typedef T type;
6  };
7
8  // Partial specialisation: recursive case if type is a pointer
9  template<typename Pointee>
10 struct RemovePointers<Pointee*>
11 {
12   typedef typename RemovePointers<Pointee>::type type;
13 };
14
15 // Partial specialisation: recursive case if type is a const pointer
16 template<typename Pointee>
17 struct RemovePointers<Pointee* const>
18 {
19   typedef typename RemovePointers<Pointee>::type type;
20 };
21
22 // Example:
23 RemovePointers<int * const * *>::type imIntVariable;
```

LISTING 2.6: *A recursive metafunction returning pointee*

recursively, stripping away further pointers, if any. On lines 16–20 another partial specialisation is used to match const pointers.

Function templates participate in normal overload resolution together with normal functions. For a particular function call, a function template is only considered if its template parameters can be deduced from the call and if this results in a syntactically valid function signature. If either of these conditions fail, overload resolution proceeds without considering the template. This *Substitution Failure Is Not An Error (SFINAE)* [Vandevoorde and Josuttis, 2003, p. 106] property of function templates is very useful, because an arbitrary compile-time computation can be placed in the return type or parameter list of a function template, and that computation can decide whether a function template can be instantiated for particular type parameters.

The SFINAE technique allows creation of C++ template metafunctions which make selections based on the existence of other functions, types, or variables. For example, a class template may use a different specialisation if its type parameter does not provide a certain member function. It can also be used to detect whether a class is derived from another class.

2.3.3 Existing template metaprogramming support libraries

Since the power of C++ template metaprogramming became widely known, generic programming and metaprogramming in C++ have become increasingly popular. Many libraries based on template metaprogramming have appeared [Veldhuizen, 1998, Veldhuizen, 2000, Alexandrescu, 2001, Landry, 2003, de Guzman and Kaiser, 2011], as well as metaprogramming libraries aimed at making template metaprogramming easier and syntactically clearer [Abrahams and Gurtovoy, 2004] (since C++ templates were not actually meant for metaprogramming, template metaprograms are often difficult to write, read and debug).

The C++ standard itself does not provide much support for metaprogramming, since metaprogramming was fairly new at the time of standardisation. However, the first library draft extension TR1 [ISO/IEC JTC1/SC22, 2006] already contains some metaprogramming facilities. They are simple metafunctions to query properties of types and to transform types in trivial ways. The new C++11 standard provides all metafunctions of TR1 with some additions, like compile-time rational arithmetic.

Currently probably the largest and most well-known metaprogramming support library is Boost Metaprogramming Library MPL [Gurtovoy and Abrahams, 2004]. It provides library components for syntactically cleaner metaprogramming, for example for type selection and compile-time arithmetic. It also contains compile-time containers, iterators, and algorithms which operate on types instead of values. Since MPL was written after the work on this thesis had already begun, metafunctions used in this thesis do not use MPL facilities. However, use of MPL would have helped if it had been available at the time.

2.3.4 Limitations of template metaprogramming

Despite its usefulness, C++ template metaprogramming has many limitations and problems which limit its use or make template metaprogramming harder. Some of these limitations affect the mechanisms presented in this thesis.

Since C++ templates were not originally meant for metaprogramming, template syntax is not particularly suited for writing metafunctions. Metafunctions are written as classes with specialisations, so the contents of the metafunction are spread to several places in the code. Since the return values of metafunctions are typedefs named by convention, code readability suffers easily. This situation can somewhat be improved by using metaprogramming libraries like Boost MPL.

One clear limitation of C++ template metaprogramming is that it is strictly a compile-time mechanism, so it is of no direct use for run-time reflection or metaprogramming. Currently **typeid** is the only run-time reflection-related mechanism in C++. However, it is possible to build some run-time mechanisms by combining the use of templates and dynamic binding.

Another limitation in template metaprogramming is that it allows creation of other types and querying properties of existing types (like the existence of a member function with a given signature), but there is no way to iterate through members of a class or struct (like in Java using its Reflection API). Similarly it is possible to check if a class is derived from another class, but there is no mechanism to iterate over all base classes of a given class. This limits usefulness of template metaprogramming for reflection, since it is not possible to navigate inheritance trees, create proxy classes with the same interface as another class, etc. Some of these limitations are discussed later in this thesis and workarounds for the limitations are presented.

Some metaprogramming techniques require the ability to use arbitrary length lists of types. However, the number of type parameters for current C++ templates is fixed by the template definition.

There are several solutions to this arbitrary length type list problem. One is to write a different template for each number of required type parameters (for example, the Boost MPL library provides template `list2` for type lists of two elements, `list3` for three, etc.). Another solution is to write one template with a large number of parameters and give each parameter a default value. Then any number of type parameters (up to the selected maximum) can be given and the rest of type parameters get their default value which can be detected by the metafunction.

A third option (used in some techniques in this thesis) is to use function types as containers for type lists of arbitrary length. A function type in C++ specifies a return type and the types of all parameters of a function. The parameter list is a list of types, so a function type can be used as a type list. For example, type **void(char, double,** `string`**)** is the type of a function with no return type and parameter types **char**, **double**, and `string`. The syntax has always been valid in C++ but is seldom used (C++ TR1 uses it for function object binders [ISO/IEC JTC1/SC22, 2006, §3.7.2][Becker, 2007, Ch. 9]). Function types can be used to pass type lists around as a single type, but decoding parameter types from the function type still requires using partial specialisation for each number of parameters.

In the C++11 standard the problem is solved by introducing *variadic templates* [ISO/IEC, 2012, §14.6.3]. Variadic templates can contain an arbitrary number of type parameters. Type parameters are represented by a single *template parameter pack*, which can be forwarded to other templates, or expanded to a list of types.

Chapter 3

Futures and active objects in the KC++ system

This chapter presents an overview of the KC++ system, a library and a precompiler for C++ which provides concurrency services using active objects, proxies, and futures. The KC++ system was the subject of the author's licentiate thesis [Rintala, 2000] and is used in this thesis as an implementation platform upon which concurrent exception support is built and tested. More detailed information on the KC++ system can be found in [Rintala, 2000].

This chapter is not considered a main contribution to this thesis, but is included to provide necessary information for the proof-of-concept implementations of the main contributions as well as for the case study. Especially it should be noted that while KC++ uses active objects for its concurrency, the contributions presented this thesis do not depend on them. Similarly, the KC++ precompiler described in this chapter is not needed for the main contributions.

3.1 The KC++ system

The KC++ system consists of a KC++-to-C++ precompiler and a class library providing run-time support for concurrent processes and their communication. KC++ programs are syntactically valid C++ programs (and vice versa), so existing C++ style analysers, statistical tools etc. can be used with KC++ programs without modifications. In this re-

spect KC++ resembles C++// [Caromel *et al.*, 1996], which also introduces concurrency without extending the C++ syntax.

KC++ is based on the UC++ language [Winder *et al.*, 1996] and many of its design goals are similar to UC++. Its aim is to add concurrency support to C++ as "naturally" as possible, allowing existing object oriented language features and C++ programming idioms to be used, without forcing the programmer to use any particular coding and design style.

Like in UC++, concurrency in KC++ is based on the *active object* concept. A concurrent KC++ program usually consists of several active objects communicating with each other. The active object model is also behind many other concurrent C++ systems, for example C++// and ABC++.

The code produced by the KC++ precompiler is close to the original KC++ code. Most modifications are only changes to type names. This means that the code produced by the precompiler is expected to be debuggable with most normal C++ debuggers. Keeping the changes minimal also makes the produced code understandable for human readers.

3.1.1 The KC++ precompiler

The current KC++ precompiler is implemented as a back end to an EDG C++ compiler front end from Edison Design Group [Edison Design Group, 2011]. The KC++ syntax is exactly the same as normal C++ syntax, so the compiler front end has not been modified in any way. The interface between the EDG front end and the KC++ back end has been kept as small as possible and it has been isolated into its own program module. This means that coupling between the front end and the back end is relatively low, making it easier to implement the KC++ precompiler using a different C++ front end, if necessary. It also makes it possible to publish most of the KC++ precompiler program code without violating non-disclosure agreements with Edison Design Group.

The compilation process is shown in Figure 3.1 on the facing page. During the compilation, the precompiler front end first compiles the KC++ program into its own intermediate code, which is represented as a graph-like data structure in the precompiler. The C++ syntax checking is performed during this phase, so only those KC++ programs which are syntactically correct C++ enter the second phase.

During the second phase, the KC++ back end scans through the intermediate code to find all the code related to active objects. It then uses the original source files

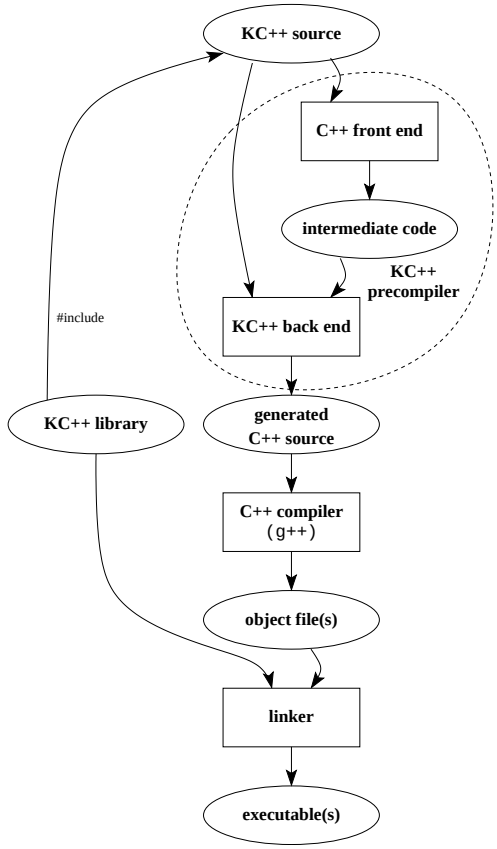

FIGURE 3.1: *The KC++ compilation process*

to create new, modified C++ source files. These files are then compiled with a conventional C++ compiler. The g++ compiler from the Gnu Compiler Collection (GCC) [GNU, 2011] has been used for this purpose, but any C++ compiler should work. The source generated by the back end is standard-conforming and uses the KC++ library to achieve concurrency.

The modifications to the original source code are kept at minimum. Using modified original source code instead of generating the code from the intermediate code makes the resulting code very close to the original. Because of this, the source code generated by the KC++ precompiler is completely human-readable and mostly understandable to programmers who are not familiar with the internals of KC++. The

precompiler generates necessary `#line` directives so that debugging tools are directed to the lines of the original source code.

In addition to minimal modifications, the KC++ precompiler creates new C++ code based on the source code. Based on active classes in the source, the precompiler creates declarations and definitions of necessary proxy classes and method invokers.

3.1.2 The KC++ library

The KC++ library is where most of the KC++ functionality resides. The KC++ precompiler mostly produces calls to the library, where actual work is done. The library includes run-time components needed to run concurrent KC++ programs as well as compile-time metaprograms responsible for most of code generation.

The code produced by the KC++ precompiler has been designed to be as minimal as possible, and most of the work is performed by template metaprograms in the KC++ library. This way it is quite possible to leave the KC++ precompiler out of the process completely, if the programmer is ready to write the simple proxy and invoker classes herself. This was considered important since the current KC++ precompiler uses the EDG C++ front-end, which is not publicly available.

The KC++ library contains classes for futures, message passing, exception handling, and exception reduction. The template code provides metafunctions for type mapping and serialisation needed in active object proxy classes and method invokers, and template classes for active object pointers and references.

3.2 Active objects

In the KC++ active object model each active object consists of a thread of execution and a data address space. The data address space contains all the data members of the object, method parameters and any data or objects which have been dynamically created inside the object. Because the address space of each active object may be distinct, active objects may not refer to any global variables or static data members.

The programmer marks objects as active by deriving their class (directly or indirectly) from a base class `Active`, which is defined in the KC++ library. All objects created from these *active classes* are automatically active objects. This is similar to the approach used in ABC++ and other “type based” active object systems, but differ-

ent from UC++ where the way the object is created defines whether an object becomes active or not.

Active objects communicate by calling each others' methods and passing data and futures to each other (KC++ futures are discussed later in Section 3.3). When an active object is created, a *proxy object* is created in the address space of the creating process. The KC++ precompiler creates a separate proxy class for each active object, and the public interface of the proxy class contains the same methods as the active class. Proxy classes also form an inheritance hierarchy following the hierarchy of active classes.

Figure 3.2 shows an active object method call in KC++. When a method of the proxy object is called, the method uses KC++ library metafunctions and classes to marshal method parameters to a method message, and sends this message to the process running the active object. There the message is parsed and a suitable *method invoker* code (again generated using KC++ library metafunctions) is called. This method invoker calls the actual method of the active object and sends its return value back by binding the return value future.

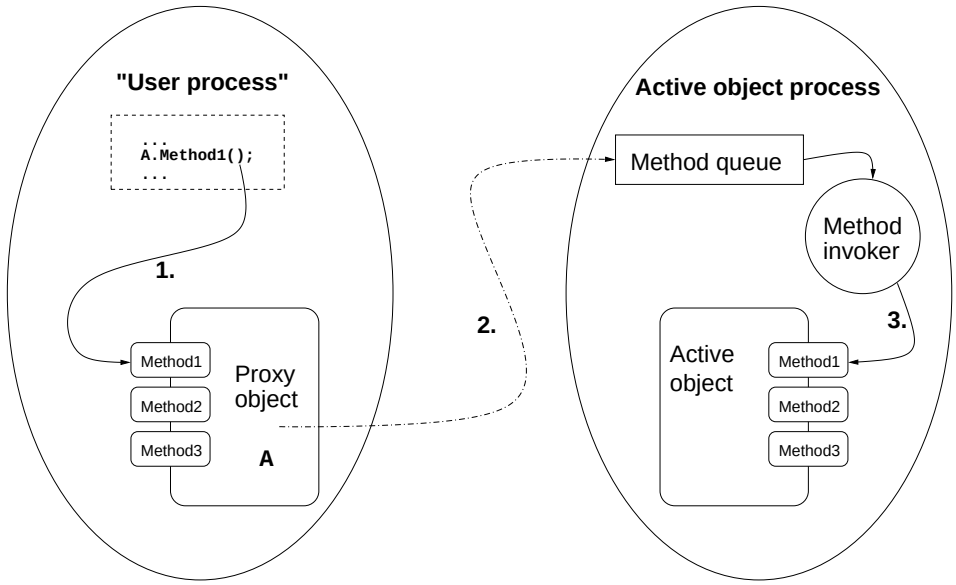

FIGURE 3.2: *Calling a method of an active object in KC++*

3.2.1 Concurrency in active objects

Each active object in KC++ contains one thread of execution. This thread receives method requests and serves them one at a time. Other method requests are collected in a queue while one method is executing. The execution of a method is non-interruptible, i.e. it follows RTC (run-to-completion) semantics, so if the execution of a method is paused by synchronisation, the whole active object is "blocked" until the method can resume its execution. Usually, method requests are served in a strict FIFO (First-In-First-Out) order, but KC++ also provides *lock objects*, which allow a user to ask for an exclusive access to an active object.

Allowing several concurrent methods in an active object would have been more flexible, but it would have involved difficult mutual exclusion and synchronisation problems. If the number of concurrently running methods is restricted to one for each active object, there are no internal mutual exclusion problems, as only one method may access the internals of an active object at any time. These kinds of active objects resemble *monitors*, a programming construct to handle mutual exclusion in concurrent programming, made widely known by Hoare [Hoare, 1974].

Even with just one method at a time, there is still one choice to be made: whether to allow a method to give way to another method and continue afterwards. This behaviour resembles the "signal" and "wait" operations in monitors, and is called "ask politely" interruption by Herb Sutter in [Sutter, 2008].

This "voluntary interruption" is useful, but it requires active objects to remember and store the state of the method execution, execute another method and then use the stored state to continue the execution of the original method. This can be implemented using multiple threads in each active object, but it increases the overhead of a method call.

The current KC++ system uses a restricted form of voluntary interruption, where an active object method can call a method `yield` in places where it is willing to let another method to execute, if necessary. If there are other methods in the method queue, one of them is executed and then execution returns from the `yield` method. The `yield` method returns a boolean telling whether there were any methods to be executed or not. An active object method can also suspend its execution until new messages have been received by calling `wait_for_messages`.

With yielding, only one thread of execution is needed for voluntary interruption. This simple implementation means that the original method can only continue after the other method has been completed, so it has its limitations.

As an alternative to yielding, KC++ active objects can also schedule their own methods to be executed when a future (or a collection of futures) becomes ready. If an active object method has to wait for futures before running to completion, waiting blocks the whole active object. Instead the method can be split into two methods, and the first method can schedule the second method to be executed when the futures receive their value. This way the active object is freed to execute other methods while the futures are still pending.

3.2.2 Creating active objects

Creation of active objects happens exactly in the same way as creating normal C++ objects. They can be created dynamically with **new**, as local variables inside a function (or any other code block), as data members of an object (or active object) or even as value parameters to a function or a member function (although this is rarely useful). This is different from most other concurrent C++ systems where active objects usually have to be created dynamically with **new** or in some language specific way. Listing 3.1 on the following page shows how active objects can be created in KC++.

The possibility to create active objects without **new** is useful for exception safety. Objects created with **new** are not automatically destroyed, so it is the responsibility of the programmer to make sure that every object (including active objects) is destroyed properly when exceptions occur. On the other hand, the C++ exception mechanism takes care of the destruction of any object whose lifetime ends because of an exception. This also holds for KC++ active objects, so all active objects which go out of scope because of an exception are automatically destroyed.

3.2.3 Active object lifetimes

Asynchronous execution of active object methods makes lifetime management of active objects more complicated than in a sequential environment. The C++ language does not provide garbage collection, so normally the lifetime of an object is determined by its scope in the program, or handled manually if it was created with **new**.

When active objects are passed as method parameters to other active objects, those methods execute asynchronously. If the caller never synchronises with the completion of the method, it becomes unclear when the parameter active object can

```
1  class MyAClass : public Active
2  {
3  public:
4     explicit MyAClass(int i);
5     MyAClass(const MyAClass& r);
6     ~MyAClass();
7     int foo1(int i);
8     MyAClass* foo2(MyAClass& a);
9     MyAClass& operator =(const MyAClass& r);
10
11 private:
12    // Private implementation details here
13 };
14
15 void func(MyAClass a) // A parameter active object
16 {
17    a.foo1(3);
18 }
19
20 int main()
21 {
22    MyAClass a1(1); // A local active object
23    int i1 = a1.foo1(3); // Synchronous
24    Future<int> i2 = a1.foo1(4); // Asynchronous
25    MyAClass* a2p = new MyAClass(2); // A dynamic active object
26    MyAClass* a3p = a1.foo2(*a2p);
27    func(a1);
28 }
```

LISTING 3.1: *An active class and creating active objects*

be destroyed, if both the caller and the method access the object. This applies both to scope-based lifetime management and manual management.

To solve the problem, KC++ uses *reference counting* [Collins, 1960, Wilson, 1992] as a simple garbage collection scheme to determine the lifetime of active objects. The active object method invoker maintains a reference count of its proxy objects. When a new proxy object is created (for example by passing the active object as a parameter to another active object), it sends a message increasing the reference count by one. Similarly, the destruction of a proxy decreases the count. When the count reaches zero, the active object is destroyed.

Reference counting has known problems with reference cycles where two reference counted objects refer to each other (of course the cycle can also be longer) [McBeth, 1963]. This is particularly problematic in object-oriented programming where bidirectional associations between objects are enough to form cycles. This problem is also known in normal C++ using dynamically created objects. The C++ ex-

tension TR1 provides tools to partially solve the problem: *strong and weak pointers* [ISO/IEC JTC1/SC22, 2006].

3.2.4 Strong and weak pointers

In C++, strong and weak pointers were originally introduced in the Boost library collection [Adler *et al.*, 2002], from where they were included in the TR1 library extension and later to the new C++11 standard. They solve the cyclic reference problem by dividing pointers to two categories: strong pointers *owning* the object and weak pointers with no ownership.

Strong pointers use normal reference counting, so an object is destroyed when the last strong pointer to it is destroyed. In C++ TR1, strong pointers are available as a template class `shared_ptr`. The semantics of strong pointers is similar to that of KC++ active object pointers, so when TR1 was introduced, KC++ was modified to support `shared_ptr`s to active objects as well.

Weak pointers do not participate in updating the reference count, so an object may be destroyed while there are weak pointers still pointing to it. When that happens, weak pointers pointing to the object become *expired*. In C++ TR1, weak pointers are provided as a template class `weak_ptr`. Their interface has a method `expired`, which can be used to check whether the object at the end of the weak pointers is still alive or not.

Since the object at the end of a weak pointer may be expired, weak pointers cannot be used to access the object directly. Instead, they provide member functions to create a strong pointer pointing to the same object, and the object can be access using that strong pointer. If the object has already been destroyed, these member functions either return a null pointer or throw an exception (depending on the member function).

The semantics of `shared_ptr` and `weak_ptr` in C++ TR1 are exactly what is needed for handling KC++ active object lifetimes. Since TR1 shared pointers are implemented as templates, KC++ can specialise them for cases where the pointee is a proxy class, and those specialisations transfer reference counting to the KC++ library instead. The specialisations also support marshalling so they can be passed from one active object to another. This way the programmer can use shared pointer tools from TR1 to handle active objects and their lifetimes.

3.3 Futures and future sources

As explained in Chapter 2, futures are placeholders for eventual return values from asynchronous calls. KC++ futures are implemented as a template class parametrised with the actual return value type. When an asynchronous call is performed, the caller immediately gets a *pending* future object representing the return value, while the call itself continues executing concurrently. When the call completes, its return value is automatically transferred to the future, where the caller can access it. If the value of the future is requested before the call completes, the future suspends the execution of the requesting thread until the value becomes available.

If return value futures were the only mechanism for synchronisation between threads of execution, all synchronisation between active objects would be restricted to the completion of method calls. A mechanism is needed to signal a process at an arbitrary time. Since pending futures can be sent among processes as parameters and return values, all that is needed is a way to create pending futures explicitly and a method to bind those futures to a given value.

3.3.1 Future sources

Future sources are such a mechanism. They are objects which can be used to explicitly create pending futures. Later the future source object can be used to give a value to all futures which have been copied from that particular future source — or any future created from these futures. Future sources make it possible to return a pending future from one call and later give it a value in another call.

Like futures, a future source is a class template with the underlying type as a type parameter. When a future source object is created, it is initially "empty," and does not contain any value. All futures created from this kind of empty future source are pending futures.

Later, a method `bind` can be used to give a future source its value. After this, all futures which have been created from this future source are bound to this value. Future forwarding mechanism is used to send the value to futures in other execution threads, if necessary. The act of binding a future source to a given value is similar to giving a return value future its value at the end of an asynchronous call.

Future sources can also be bound to exceptions, thus propagating that exception as a "value" of generated futures. Pending futures can also be used to bind a future

source. In this case, the future source is bound to the eventual value of the future when it becomes available.

3.3.2 An example with futures

Listing 3.2 on page 45 shows a simple code example with asynchronous calls. Methods `asyncCall1`, `asyncCall2`, and `setValue` of active class `Server` are executed asynchronously. Active objects of class `Client1` and `Client2` also execute their methods in their own threads. Figure 3.3 on the following page shows a UML sequence diagram of one possible execution sequence. Messages in the diagram are marked in Listing 3.2 with numbered comments. Continuous arrows denote calls. A filled arrowhead marks a synchronous call, a stick arrowhead an asynchronous call. Where necessary, dashed arrows denote return values.

With message 3 in Figure 3.3, `Client1` makes an asynchronous call, which continues its execution concurrently with the caller. When the call completes, it sends the return value to the future `f1` (message 4), from which the caller later retrieves it (msgs 9–10).

Normally, a future gets its value automatically when an asynchronous call completes, but this is restrictive in some cases. In some programs it is practical to delay the future even further, and leave the return value future empty when the call completes. The value for the future is provided later by another call (likely called by a different thread), which binds the future to a value and thus releases the thread waiting for the future. This possibility makes futures even more useful for explicit synchronisation and signalling. Futures are well suited for such a use, because they also allow data to be passed between threads.

In Figure 3.3 on the next page, messages 5–8 and 11-15 show an example of future sources. Method `asyncCall2` gets its return value future from the future source `fs`. Initially, this future source contains no value and futures generated from it are empty. Thus future `f2` stays pending after the asynchronous call completes (msgs 6–8). Later a thread running `caller2` calls a function which binds the future source to a value. This value is automatically propagated to `f2`, and `caller1` waiting for its value is released (msgs 12–15).

The combination of future sources and futures is somewhat similar to *barriers* [Andrews, 1991, Ch. 4]. However, synchronisation with futures sources can be triggered at any time, whereas barriers usually require a predetermined number

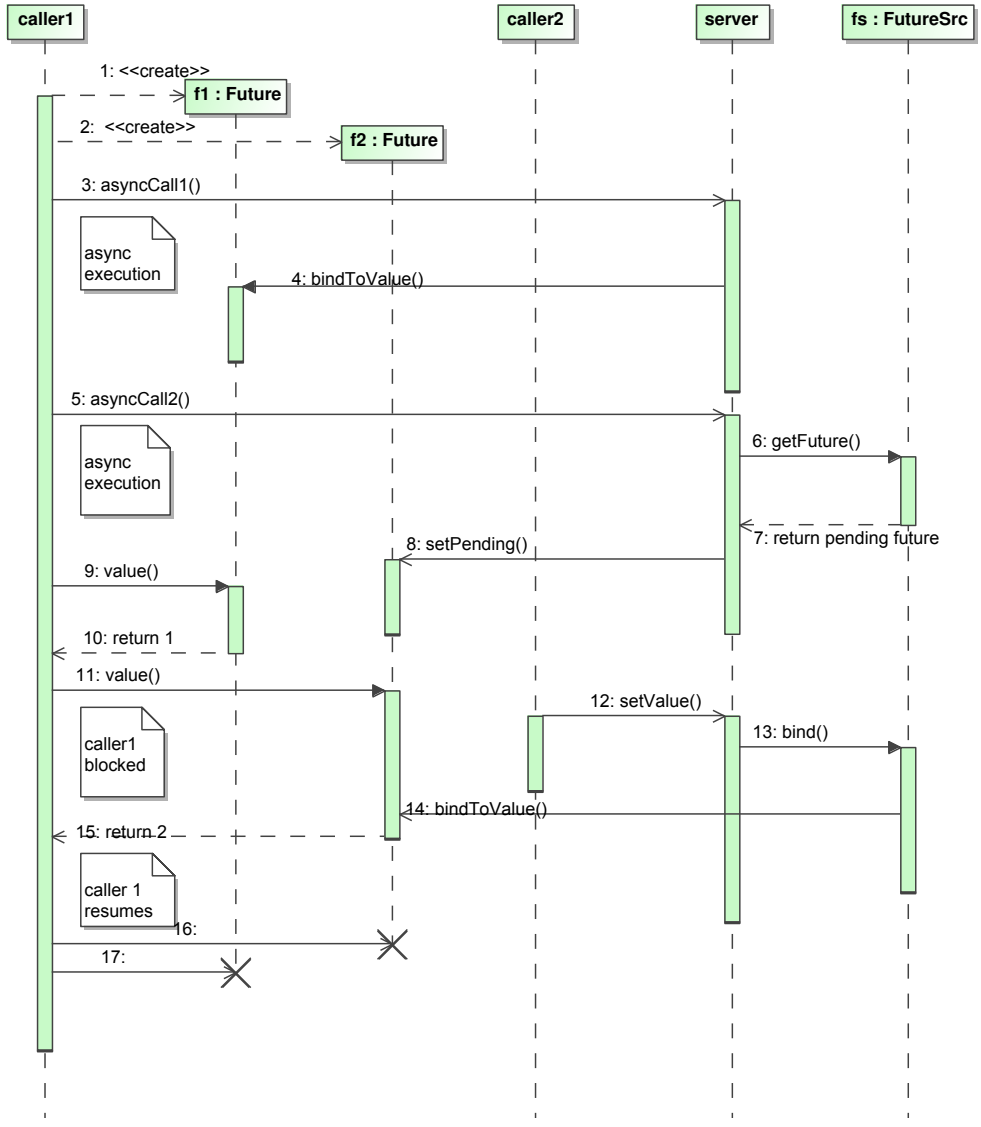

FIGURE 3.3: *Sequence diagram of Listing 3.2*

```
1  class Server : public Active
2  {
3  public:
4     Future<int> asyncCall1() { return 1; } // 4.
5     Future<int> asyncCall2() { return fs.getFuture(); } // 6.-8.
6     void setValue(int i) { fs.bind(i); } // 13.-14.
7
8  private:
9     FutureSrc<int> fs;
10 };
11
12 class Caller1 : public Active
13 {
14 public:
15    void method(std::tr1::shared_ptr<Server> server)
16    {
17       Future<int> f1, f2; // 1.-2.
18       f1 = server->asyncCall1(); // 3.
19       f2 = server->asyncCall2(); // 5.
20       cout << f1.value(); // 9.-10.
21       cout << f2.value(); // 11. & 15.
22    } // 16.-17.
23 };
24
25 class Caller2 : public Active
26 {
27 public:
28    void method(std::tr1::shared_ptr<Server> server)
29    {
30       server->setValue(2); // 12.
31    }
32 };
```

LISTING 3.2: *Asynchronous calls using futures*

of participants. In addition to this, future sources allow data to be passed from the synchroniser to released threads.

3.3.3 Destroying futures and future sources

When a future is destroyed, its possible value (or exception) is destroyed with it. A future can be destroyed before it receives its value, in which case the value is still received by the KC++ library, but it is destroyed immediately (this makes sure possible side effects of creating the value happen also in this case). If a future has been copied to other futures before, and is destroyed before receiving its value, that value is still propagated to copied futures by the KC++ library.

Destruction of future sources needs also special attention. If a future source has already been bound to a value (or exception), its destruction is trivial. However, destroying an unbound future source is dangerous, since futures may be waiting for its value. Since it is not possible to prevent the destruction of a future source, a logical solution in this case is to bind the future source to an exception (`FutureSrcNotBound` in KC++) before its destruction, and send the exception to waiting futures. This prevents deadlocks even if a future source is destroyed prematurely.

The semantics of destroying futures and future sources ensure that all futures eventually receive a value (assuming all threads of execution eventually terminate), and that destruction of a future or a future source does not cause deadlocks even if they are destroyed before receiving a value.

3.3.4 Pure synchronisation — void-futures

Although futures are normally used as placeholders for return values of asynchronous calls, they have another important property — they allow callers to synchronise with the completion of a call. Synchronisation in general is an issue separate from return value passing (although the act of receiving a return value always includes synchronisation).

Despite being originally designed as placeholders for a return value, futures can also be used for pure synchronisation. Futures with **`void`** as the underlying type are a special case, implemented as a specialisation of the future template. They contain the necessary mechanisms for synchronisation, but no underlying value or member functions for accessing it.

A function returning a void-future can be called like a function returning nothing. In this case, the call is asynchronous and the returned void-future is discarded. If synchronisation is wanted, the caller can synchronise with the returned void-future. Like with normal futures, the third possibility is to occasionally poll the void-future with `isready` and perform some other actions while waiting.

In addition to synchronisation with calls with no return value, void-futures are very useful when combined with future sources. They can be used as explicit synchronisation tokens which can be stored in containers, passed between execution threads etc. They also make the concurrency model simpler because futures can be used as the only synchronisation mechanism in the whole system.

3.4 Passing parameters to active objects

KC++ active objects do not have to exist in the same address space. This means that normal C++ parameter passing mechanism cannot be used to pass method parameters to active objects, because C++ parameters require a shared address space.

Active objects communicate with each other using the KC++ message passing subsystem. All parameters have to be *marshalled* into a data stream which is passed as a part of the method request. A method request message consists of the method identifier (identifying the requested method) and a data stream which contains all the marshalled parameters. When an active object receives the method request, it first identifies the method (using the identifier) and then unmarshals the data stream, creating local copies of the parameters.

The KC++ marshalling/unmarshalling system resembles the C++// marshalling system or the CORBA GIOP protocol [OMG, 1998, Ch. 13]. However, KC++ does not store any type information (like C++// meta-classes) during marshalling, so a marshalled parameter list contains just marshalled parameter values concatenated one after another. Storing type information is not needed in a language like C++ because the method signature already fixes the number and types of the parameters. If each different version of an overloaded method is considered a different method with a different method ID, the method ID already contains all necessary information about parameter types.

The KC++ library contains functions to marshal and unmarshal all C++ built-in types except for pointers and references. This makes passing parameters of built-in types to active objects identical to passing them as parameters to normal objects.

If the programmer wants to pass her own data types (data structures, non-active objects etc.), she must write appropriate marshalling and unmarshalling functions for them. The KC++ library provides classes `OMsg` (output message) and `IMsg` (input message), which represent data streams to and from which data is to be marshalled to using operators << and >>. These can be overloaded for user-defined data types.

The only exceptions to this are pointers and references to normal objects. They cannot be passed as parameters at all, because they require shared memory.

Unlike with normal pointers, there are no restrictions on using active object pointers or references as parameters to active object methods. Active objects already have a possibility to refer to each other through proxies. In a sense, all active objects occupy the same “active object address space,” so passing active object pointers and

references between active objects poses no problems. Both strong and weak pointers can be passed as parameters, as well as normal C++ pointers and references to active objects (which act as strong pointers for object lifetime management).

KC++ future types are also allowed as parameters to active object methods. The only requirement is that the underlying type of the future type must also be valid as a parameter. If a future passed to a method has already received its value, the value contained in the future is marshalled and passed in the method message. Alternatively, if a pending future is passed, the id of the future is passed in the message. Later, when the value of the future becomes known, the KC++ run-time system takes care of forwarding the value to other active objects which contain copies of the future.

Chapter 4

RPC exceptions using C++ template metaprogramming

Serialisation of data is needed when information is passed among program entities that have no shared memory. For remote procedure calls, this includes serialisation of exception objects in addition to more traditional parameters and return values. In C++, serialisation of exceptions is more complicated than parameters and return values, since internal copying and passing of exceptions is handled differently from C++ parameters and return values. This chapter presents a light-weight template metaprogramming based mechanism for passing C++ exceptions in remote procedure calls (RPC), remote method invocations (RMI), and other situations where the caller and the callee do not have a shared address space.

Basic ideas in this chapter were originally published in [Rintala, 2007], but have been since extended and developed further. The mechanisms in this chapter do not depend on any specific model or implementation of concurrency in C++. They can be used in any context requiring serialisation of exceptions.

4.1 Introduction

In many programs, it is quite common to have one process (or some other entity) acting as a server, serving requests from clients. If the server and client do not have shared memory, serialisation of data is needed when the caller and the callee

communicate with each other, often through some sort of message passing system. This need may arise in a distributed environment using RPC (remote procedure call) or RMI (remote method invocation), or in a parallel or concurrent system where the server and the client are separate processes without shared memory. For the rest of this chapter, the abbreviation "RPC" is used loosely to refer to all these situations. Serialisation is included in many programming languages like Java, and is a part of many language independent systems like CORBA, but the C++ language has no built-in support for serialisation.

Exceptions are rapidly becoming the most common error handling mechanism, which means serialisation mechanisms in RPC should also be able to pass exceptions in addition to parameters and return values. This functionality is still missing from many RPC libraries, forcing programmers to revert to traditional return values for error handling.

Standard C++ passes objects as parameters and return values differently from many other object-oriented languages. In C++, objects are passed by copying them based on the static type defined in the type signature of the function, rather than cloning the object based on its dynamic type (Java RMI) or sharing objects without copying them (normal Java). Explicit sharing is of course possible in C++ through pointer and reference types.

Copying objects based on their static type allows C++ to optimise the performance of parameter and return value passing when objects are small. It also makes serialisation easier in most cases. However, exception handling in C++ differs from parameters and return values in this respect. C++ exception objects are copied based on their real dynamic type, making serialisation of exceptions more complex, when they are passed across address space boundaries (the interface has no compile-time knowledge on the type of the exception object).

This chapter analyses how exception handling in C++ affects serialisation in remote procedure calls. A light-weight template-based solution solving these problems is then presented, and its performance and ease of use is analysed.

4.2 Problem analysis

From the interface point of view, exceptions are in a certain sense an alternative to return values. They also transfer information to the caller, i.e. the type of the exception and possibly data embedded in the exception object. Serialisation services are

needed for parameter and return values passing. Similarly, serialisation is needed for exception support. Exceptions must be handled for both synchronous and asynchronous calls. When these calls occur across address space boundaries, implementing exception propagation becomes significantly more complex. This chapter addresses exception propagation across address space boundaries.

In addition to RPC, serialisation is routinely needed for persistence services. However, persistence normally deals only with the state of objects. Therefore, exceptions (which are mainly control-flow signals) are not an issue with serialisation libraries aimed for persistence.

The C++ language does not provide support for serialisation. Distributed systems like CORBA solve this problem by providing their own serialisation libraries. However, custom code is needed for serialisation of user-defined classes and data types. Complex frameworks like CORBA offer generality and let systems communicate with each other over machine architecture and programming language boundaries.

However, for many light-weight applications, such frameworks are unnecessarily heavyweight and complex, and they require extra programmer effort in the form of IDL specifications, etc. For example, a program may just need a few simple distributed remote procedure calls, or it may consist of a few concurrent processes communicating with each other on the same machine. Such applications tend to use lighter weight serialisation libraries, usually without separate code generator programs.

As mentioned, C++ *copies* objects when they are passed as parameters or return values. This differs from many other object-oriented languages like Java, where all objects are passed by reference. However, the copying approach is used in most languages for passing basic types (integers, etc.) by value instead of by reference. The copy semantics is also ideal for RPC and serialisation, since sharing an object is difficult in calls across address spaces and computers (even Java RMI copies its parameters). Similarly, copying makes object lifetime management easier on the language level, since the destruction of each copy is determined by its location in the program, which is essential in C++ where garbage-collection is not built into the language.

When objects are copied in C++, the type of the copy is the *static* type used to reference the original object, i.e. in this case the type of the parameter or return value. With inheritance, this may end up *slicing* the object when the dynamic type of the object differs from the type used in copying [Budd, 2002, Ch. 27], which

causes problems with C++ exceptions. These problems are discussed later in this section.

Pointer and reference parameters are usually used to pass objects whose dynamic type may differ, circumventing the slicing problem. However, pointer and reference passing is difficult to implement without shared memory. For these reasons pointer and reference parameters are not usually allowed as parameters for RPC and are not discussed further in this chapter.

Copying based on static typing in combination with templates (which also use compile-time static typing) makes it quite straightforward to implement libraries for marshalling parameters and return values into data messages, sending these messages to the receiver, and then creating and unmarshalling those parameters and return values from the message. Template-based serialisation libraries already exist for C++ [Bartosik, 2004]. However, when exception handling is added to the picture, the situation is different.

When an exception is thrown, a copy is made of the exception object, based on the static type used in the **throw** statement. This semantics means it is *not* possible to properly throw an exception whose real dynamic type is not the same as the static type used to throw it. As an example, if an exception is caught with a catch clause `catch (const E& e)`, it is not possible to reliably re-throw the exception with `throw e`. If the type of the original exception is a subclass of `E`, the throw statement throws a sliced copy. This is the main reason why re-throwing in C++ is allowed only using special syntax `throw;`, which re-throws the original exception object without copying it.

When a copy of an exception object is propagated out of a function call, its type is *not* statically defined by the type signature of the function. Exception specifications allow functions to declare *base classes* of allowed exceptions, but the dynamic type of exception objects can be any derived class. This means that information about the dynamic type of an exception object is needed when it is sent to a different address space. Similarly, the receiver of an exception object (the caller) has to be able to re-create the exception without static compile-time knowledge of its type. Finally, the re-created exception object has to be thrown, again without knowing its type at compile-time.

All this makes passing exception objects across address spaces more difficult in C++ than passing parameters and return values. Especially it makes the use of static template metaprogramming more challenging. The following sections analyse the

problem further. A template-based solution using automatically generated dynamic factories is then described.

4.3 Server side — catching and marshalling

As mentioned, propagating an exception back to the caller in a distributed environment requires copying the exception object between address spaces. This requirement does not cause incompatibilities with normal C++ semantics, since the language explicitly states that copying of exception objects may occur even in normal C++ [ISO/IEC, 2003, §15.1/3]. However, the exception propagation mechanism is the only place in C++ where the dynamic type of a compiler-generated copy (a thrown exception) may differ from the static type used to destroy the object (in the exception handler).

Figure 4.1 shows a typical call sequence that ends in an exception. When a call request is sent to a server, its invoker code is responsible for interpreting the data in the request and then calling an appropriate C++ function that actually executes the requested service. If an exception is thrown within the function, it is the responsibility of the invoker to catch it and send it to the calling client. This requirement means that the invoker has to be able to catch and handle different types of exceptions. In the figure, notes marked with an asterisk (*) denote places where knowing the dynamic type of the exception object is required.

The normal way to catch all exceptions in C++ is to use the **`catch (...)`** syntax. However, this mechanism is of no use here, because it does not give the error handling code any way to access the thrown exception object. Even the type of the thrown exception is not known to the exception handler.

The only other way for the function invoker code to catch exceptions is to require that all such exceptions are derived from a common base class (this is a usual requirement in many other object-oriented programming languages). The solution described in this chapter requires that exceptions thrown across address spaces have to be derived from a base class `RpcExcpBase`, which allows the derivation of methods for serialisation, dynamic creation of exception objects, etc. Derivation from this class happens through an intermediate base class for template metaprogramming reasons and is explained later.

After the server has caught an exception using the common base class, it must be able to marshal the exception into a data stream and send this stream back to

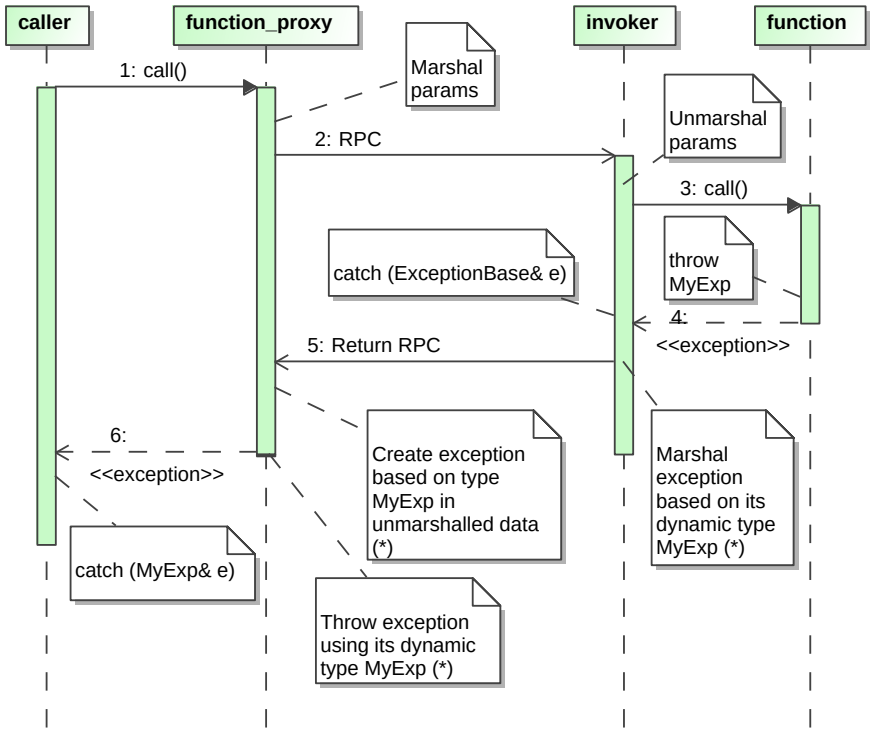

FIGURE 4.1: *Calling an RPC function and throwing an exception*

the caller as part of the return value message. Marshalling is implemented with a pure virtual function `marshal` in the exception base class. The invoker calls this member function and passes as a parameter the data stream, which is used to send the message.

The structure of RPC exception classes is shown as a UML class diagram in Figure 4.2 on the facing page. In addition to a common base class `RpcExcpBase`, an intermediate exception base class template `RpcException` is provided. The programmer is expected to derive all RPC exceptions from this template, and give the actual exception class *as a template parameter* to the base class template (as indicated with the ≪ bind ≫ stereotype in the diagram), e.g.:

```
class MyException : public RpcException<MyException>
{ /* Normal exception class definition */ };
```

This idiom is often called the *Curiously Recurring Template Pattern* or CRTP [Coplien, 1995, Vandevoorde and Josuttis, 2003, §16.3]. Use of CRTP means that each derived exception class has its own unique instance of the base class template, and this base class instance *statically* knows the type of the derived class it belongs to.[1]

The instance of the base class template statically knows the type of the derived exception objects through the template parameter. Therefore it can implement the necessary virtual functions of `RpcExcpBase` (like `marshal`), releasing the programmer of `MyException` from that duty.

4.4 Client side — unmarshalling and re-throwing

C++ is a statically typed language with very limited run-time reflective capabilities. This limitation means that when an object is created (including throwing an exception), the type of the object has to be known at compile time [ISO/IEC, 2003, §15.1/3].

However, the creation of an RPC exception object on the client side has to be performed based on the dynamic type stored in the received message, which is inherently a run-time issue. Therefore, appropriate mechanisms for limited run-

[1] CRTP is possible because although the definition of the derived class is incomplete at the time when the base class template must be instantiated, the incomplete type is enough to instantiate only the *definition* of the base class template [ISO/IEC, 2003, §14.7.1/1]. The implementations of the member functions are not usually instantiated at this point [Vandevoorde and Josuttis, 2003, §10.2].

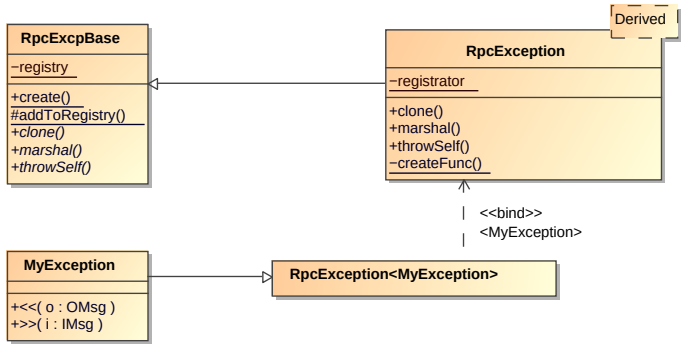

FIGURE 4.2: *Structure of the RPC exception mechanism*

time reflection have to be built. The client-side mechanisms are more complex than those on the server side, where the template mechanisms just make it easier to write exception classes.

4.4.1 Unmarshalling exception objects

C++ has basic RTTI (Run-Time Type Information) constructs **`dynamic_cast`** and **`typeid`** [ISO/IEC, 2003, §5.2.7&8], but these cannot be used to create objects, only to *ask* about the dynamic type of an object. Therefore, the client side mechanisms for creating exception objects from received data must be written from scratch.

The solution in this chapter is based on the Abstract Factory design pattern [Gamma *et al.*, 1996, pp. 87–95], which is automated using the template-based inheritance described earlier. The exception base class `RpcExcpBase` keeps a static data member `registry`, a data structure that maps the type id of each exception class to an appropriate creation function, which in turn is able to create an object of the correct type. The mapping between exception classes and type ids is discussed later in Section 4.7. That section also analyses concurrency issues concerning the registry.

The base class also has a static member function `create`, which is used by the rest of the library to dynamically create and unmarshal exception objects when needed. When the client side receives a message containing an exception, `create` chooses the correct creation function from the registry based on the type id stored in the message. The chosen creation function then creates an exception object of appropriate type and unmarshals the object using the rest of the message data. If the message contains an exception object that is not found in the registry, the system returns a generic exception `UnknownRpcException` (this possibility is discussed in more detail in the next section).

4.4.2 Automating the creation of polymorphic factory

Normally, the use of a polymorphic factory requires exception classes to provide their own creation functions. [OMG, 2003, §1.17.10] However, the CRTP-based template inheritance described earlier can also be used to automatically construct a creation function for each exception class. This approach makes it possible to embed necessary mechanisms into an otherwise normal exception class, which can be seen from Figure 4.2.

Actual creation functions are located in the CRTP class template `RpcException`. It has a private static function `createFunc` (Listing 4.1), which creates an exception object whose type is that of the *template parameter*, i.e. the actual exception object. Unmarshalling of data by the client is implemented using the default constructor and unmarshalling operator >>. For simplicity, the function assumes that creating and unmarshalling an exception object never fails (it would of course be possible to return a different exception in such a case).

Using the `RpcException` template makes it possible to automate the whole exception object creation and unmarshalling process. There is one `RpcException` instance for each actual exception class, this instance contains a creation function needed to create such objects, and the creation function knows the static type of the object to be created.

The system has to make sure that each `RpcException` instance registers its creation function to the base class `RpcExcpBase`. The simplest possibility would be to list all creation functions in a compile-time list, which could be used to initialise the polymorphic factory. However, this approach would not be practical, since it would introduce a place in the program where knowledge on all RPC exceptions would have to be gathered. Since exceptions are often defined on module or class basis, having to update a separate list of RPC exceptions would be tedious and prone to errors. An alternative solution would be to initialise the polymorphic factory during program startup, and let each RPC exception class (or rather its creation function) be registered separately. This approach still requires manual registration, leaving the system vulnerable to missing registrations.

Fortunately, the registration of each RPC exception class can be automated using templates. The system has to make sure that each `RpcException` instance registers its creation function to the base class `RpcExcpBase`. The base class provides a static protected function `addToRegistry`, which is used to add all creation functions to `registry`. The registration is handled using an intermediate class template

```
1 template <typename E>
2 RpcExcpBase* RpcException<E>::createFunc(IMsg& s) {
3   E* e = new E; s >> *e; // Unmarshal message data into the object
4   return e;
5 }
```

LISTING 4.1: *static RPC exception creation function*

`Registrator` nested inside the `RpcException` template. This class template is otherwise empty, but has a default constructor to register the creation function of the enclosing class.

Each `RpcException` template instance has a `Registrator` object as a static data member. When this data member is created during program startup, its constructor is executed and registers the `RpcException` instance. However, the C++ template instantiation mechanism is partly based on lazy evaluation, and the consequences of this are discussed in the next section.

By using registrators, the system automatically registers all exception classes declared anywhere in the code. The only requirement is that the code of the client has to contain declarations for all exception classes the server may throw (this is of course the case if both the server and the client run the same executable). This includes the declarations of thrown derived exceptions that the client catches by a base class reference. If the requirement is violated and the server throws an exception which is not declared in the client, that exception is replaced by a generic exception object `UnknownRpcException`.

4.4.3 Template instantiation issues

In C++, static members of a class template are only instantiated if they are referenced in the program [ISO/IEC, 2003, §14.7.1]. This "laziness" is a useful feature, but in this case, the static `Registrator` data member is not referred to by any part of the program, since its sole purpose is to execute its constructor when the program starts. This means that registrators would normally never be instantiated at all.

The problem can be solved by using the fact that the *type* of a static member is always instantiated even if the member itself is not. The `RpcException` class template declares an additional empty template `ForceInstance`, which requires a pointer to a `Registrator` as a template parameter. Pointer `ForceInstance<&registrator>*` is then declared as a static data member of the `RpcException` template, as shown in Listing 4.2 on the facing page.

The static data member `notEverInstantiated` is never referred to, so it is not instantiated and does not consume any memory or produce any code. However, instantiating its type in the class definition requires taking the address of the static data member `registrator`, which causes `registrator` to be instantiated.

```
1 template <typename MyException>
2 class RpcException
3 {
4   static Registrator registrator;
5   // Statically force instantiation of registrator
6   template <Registrator*> class ForceInstance;
7   static ForceInstance<&registrator>* notEverInstantiated;
8 };
```

LISTING 4.2: *Forcing instantiation of registrator*

This template metaprogramming mechanism makes the `RpcException` class template act as a program generator [Czarnecki and Eisenecker, 2000], where just referring to it (instantiating it through inheritance) automatically generates necessary factory functions and registers them to the object factory.

4.4.4 Throwing the created exception object

When an exception is re-created on the client side using the polymorphic factory in `RpcExcpBase`, an exception object is created and an `RpcExcpBase*` base class pointer is returned. Now this object has to be thrown. Just like creating an object, throwing an exception in C++ requires static compile-time knowledge of its type [ISO/IEC, 2003, §15.1/3].

`RpcExcpBase` declares a pure virtual function `throwSelf`. Its implementation in the `RpcException` class template simply downcasts `*this` to the actual exception class type (the real dynamic type of the object) and then throws the exception object itself. Using these mechanisms, the client code can create a received exception object using `create` in `RpcExcpBase` and throw the exception object by calling its `throwSelf`. Throwing the object copies it automatically, so the original exception object can be destroyed (preferably automatically during stack unwinding). Alternatively, the created exception object can be stored for throwing it later. This kind of delayed throwing is necessary in an asynchronous environment using futures for delayed return value passing (discussed in Chapter 5).

4.5 Exception hierarchies

In the exception propagation mechanisms described in this thesis, each concrete exception class is derived from a separate instance of the `RpcException` template.

Thus it is possible to generate all type-based exception handling code automatically during compile time. However, it also causes an unfortunate side effect that RPC exception classes derived from the `RpcException` class template should not be used as base classes themselves.

This restriction comes from the fact that the `RpcException` instance has to get the type of the derived exception class as its template parameter. If further derivation were allowed, this would not be true for derived classes. For this reason each RPC exception class has to be a leaf class in the inheritance hierarchy.

Exception hierarchies are extremely important in exception handling, so allowing them is essential. The problem can be solved by using multiple inheritance. The structure of the solution is shown in Figure 4.3 on the next page. The gray oval shows a "local" exception hierarchy, which is not aware of RPC issues. Actual RPC exception classes are then derived from *both* the local exception class *and* the `RpcException` template instance. This is shown with classes `E1Rpc` and `E2Rpc`. Class `E3Rpc` is an example of a class which is known to be a leaf class, so it can be added directly to the local hierarchy and does not need an intermediate base class.

Using this kind of hierarchy is quite straightforward. All exception handlers can catch exceptions from the "local" exception hierarchy, so existing code does not necessarily have to be updated. Code whose exceptions cannot end up propagating to other address spaces can also throw these exceptions. Throwing exception objects derived from `RpcExcpBase` is only necessary in places where it is possible that an exception is sent to another address space. These objects are instances of their base classes, so they can also be caught by exception handlers unaware of RPC issues:

```
try {
  int result = activeObject.remoteCall(); // May throw
} catch (const E1L& e) { /* Also catches E1Rpc */ }
```

Generation of RPC exception classes from normal exception classes can be automated using templates. Class `E4L` and template `ConcExcp<E4L>` show this in Figure 4.3. Template `ConcExcp` is otherwise empty, but it is derived both from its template parameter and the `RpcException` template instance. The result is a mixin-like class, which implements the original exception class and contains necessary mechanisms for cross-RPC propagation, shown in Listing 4.3 on the facing page.

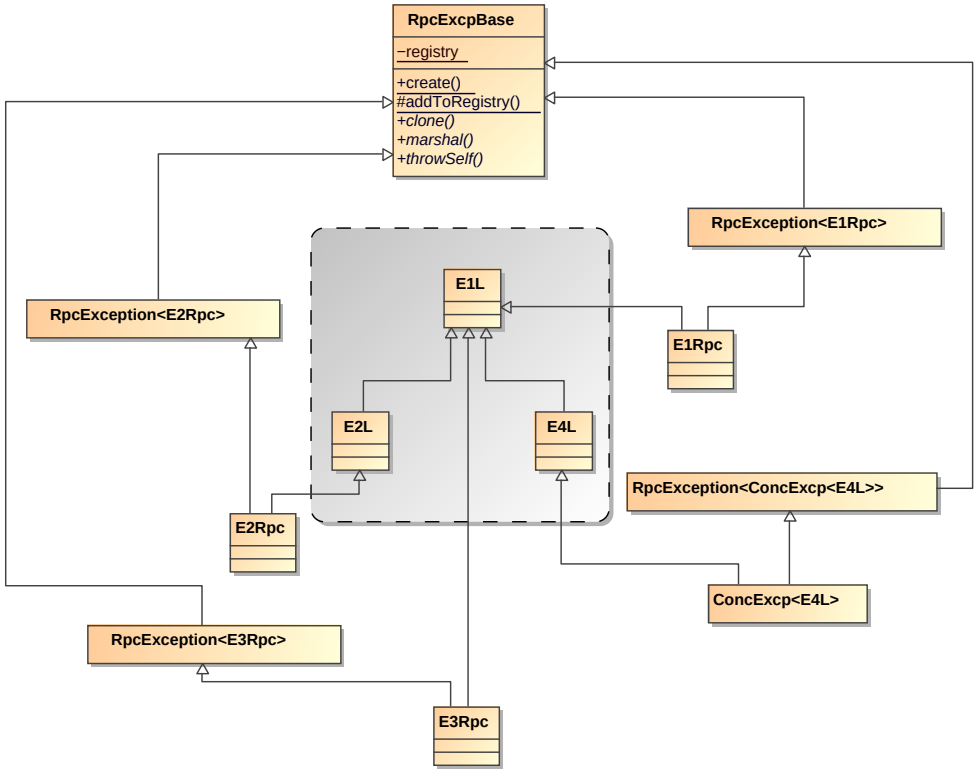

FIGURE 4.3: *Using an exception hierarchy with RPC exceptions*

```
1 template <typename LocalExcp>
2 class ConcExcp :
3   public LocalExcp,
4   public RpcException< ConcExcp<LocalExcp> >
5 {
6   /* Template pass-through constructors */
7 };
```

LISTING 4.3: *Definition of `ConcExcp`*

The ConcExcp template provides template constructors that simply call appropriate constructors of the local exception class. This way ConcExcp<E> can act as an RPC counterpart of an exception class E, and ConcExcp<E1L> could have been used directly in place of E1Rpc. Of course the programmer still has to write necessary marshalling and unmarshalling operators for the local exception classes to make serialisation possible. The ConcExcp template is enough to create the rest of mechanisms for passing exceptions over RPC boundaries.

4.6 Automatic mapping of non-RPC exceptions

The ConcExcp template can also be used to catch original exceptions and re-throw them in RPC form. The following works if the type of the catch clause parameter is the same as the real dynamic type of the thrown exception object. Otherwise copying the object ends up slicing it:

```
try { /* Code that throws an exception from "local" hierarchy */ }
catch (const E4L& obj) { throw ConcExcp<E4L>(obj); }
```

However, this kind of mapping from non-RPC exceptions to their RPC counterparts requires the programmer to provide such try-catch blocks to all places where non-RPC exceptions might escape to other processes. This is error prone and requires lots of extra code, especially since the mechanism requires a catch clause for every possible exception class to be mapped.

For these reasons a more automated solution is presented. The C++ template mechanisms combined with run-time type information make it possible to perform limited mapping in the method invoker code without requiring extra application code. The next section presents this mechanism, discusses its limitations, and explains how these limitations can be solved by requiring extra information from the programmer.

4.6.1 Direct non-polymorphic mapping of thrown exceptions

The C++ language provides limited run-time type information through the **typeid** operator. When this operator is applied to an object, it returns a type_info object representing the dynamic type of the operand. However, these type_info objects

can only be compared with each other for equality. Two `type_info` objects compare equal if the objects from which they were generated have the same dynamic type.

This mechanism makes it possible to create a registry of non-RPC exception classes for which there is an RPC counterpart. Creation of this registry can be done in the `RpcException` template using the same mechanism which is responsible for creating the RPC exception unmarshalling registry (Section 4.4.2).

When a non-RPC exception is thrown out from an RPC method, the method invoker catches it. The invoker then uses the mapping registry to find out whether the exception can be mapped to an RPC counterpart. If such a mapping is found, the RPC version of the exception is propagated to the caller.

It is only required that non-RPC exceptions are derived from `std::exception`. This requirement exists because the method invoker must be able to catch the exception, and this is impossible without a common base class of the thrown exceptions.

The only piece of information missing in the `RpcException` template is the type of the original non-RPC exception. The RPC exception class derived from the original non-RPC exception is passed to `RpcException` as a template parameter. However, C++ provides no reflection mechanisms for traversing the inheritance hierarchy to the base class. This missing base class information must be provided by the programmer in the form of a typedef, as shown in Listing 4.4 (the parenthesis at the end of typedef on line 4 are explained later).

When `RpcException` is instantiated, its registration metafunction checks whether `MapBases` typedef is present. If it is, a mapping from `E1L` to `E1Rpc` is registered. This mapping of course requires that `E1Rpc` has a constructor accepting the original `E1L` object. The mapping is implemented by storing the `type_info` of `E1L` and a pointer to a function which creates an `E1Rpc` object from an `E1L` object. This function is

```
1 class E1Rpc : public E1L, public RpcException<E1Rpc>
2 {
3 public:
4    typedef E1L MapBases(); // Provide mapping information
5    E1Rpc(const E1L& e);
          ⋮
6 };
```

LISTING 4.4: *Providing exception mapping information*

automatically created by the registration metafunction. If no MapBases typedef is present, no mapping is registered.

4.6.2 Automatic mapping information through ConcExcp

When the ConcExcp template is used to create RPC versions of non-RPC functions, the template can automatically provide the MapBases typedef. This means that RPC exceptions created with ConcExcp can automatically register the non-RPC-to-RPC exception mapping without help from the programmer. Since ConcExcp provides template constructors for all constructors of the original exception class, a normal copy constructor is enough for mapping from E to ConcExcp<E>. This means that to provide exception mapping for exception X, only marshalling operators and a typedef for the ConcExcp counterpart is needed, as shown in Listing 4.5.

Since E1L does not have to be modified, this mapping can be used for exception classes coming from a third party library, for C++ standard exceptions, etc.

However, lazy instantiation of C++ templates reveals another problem. According to the C++ standard, a class template is only instantiated when "... the class type is used in a context that requires a completely-defined object type or if the completeness of the class type affects the semantics of the program." [ISO/IEC, 2003, §14.7.1 4].

For RpcException, this is not a problem since it is used as a base class, which has to be a "completely-defined object type". However, a typedef does not require the source type to be completely defined. This means that if ConcExcp is only used in a typedef like in the earlier example, the compiler does not instantiate it at all. This in turn means that registration functions in its base RpcException do not get instantiated either.

The problem can be solved by explicitly instantiating RpcException in the template parameter list of ConcExcp. This can be achieved by adding an extra dummy template parameter to ConcExcp, and giving it as a default value a type defined in-

```
1 class E1L { /* Definition */ };
2 OMsg& operator<<(OMsg& omsg, const E1L&) { /* Marshalling */ }
3 IMsg& operator>>(IMsg& imsg, E1L&) { /* Unmarshalling */ }
4
5 typedef ConcExcp<E1L> E1Rpc;
```

LISTING 4.5: *Automatic RPC exception mapping using ConcExcp*

side RpcException (d_void is used, which is simply a typedef for **void**). When this default value is used, it forces the instantiation of RpcException and its registration mechanism. The beginning of ConcExcp definition is given in Listing 4.6.

With this addition, a ConcExcp typedef and marshalling operators are all that is needed to provide automatic mapping from non-RPC exceptions to their RPC counterparts, and this mapping is automatically used if a non-RPC exception escapes an active object method.

4.6.3 Limitations of the mapping mechanism

The mapping mechanism described above works well when exact types of possible exceptions are known. However, in object-oriented exception handling it is common to throw derived class exceptions and catch them using their base class. This is crucial since it allows exception handlers to choose the family of exceptions they are able to handle. In third party libraries it is common that only a part of the exception hierarchy is revealed to the library user, i.e. the library throws internal derived exception objects which *have* to be caught by using their base class.

The described exception mapping mechanism uses the C++ **typeid** operator and type_info objects to query the type of the exception and create a suitable RPC counterpart exception. Unfortunately type_info objects do not contain any information on inheritance relationships. This means that if an RPC counterpart has been registered for a base class exception but not for derived classes, the mechanism cannot map thrown derived class exceptions to their base class RPC counterparts, preventing those derived class exceptions to be propagated to a different process.

To solve this problem, information about the inheritance hierarchy must be provided. Since C++ lacks reflection capabilities to provide such information, it must be given by the programmer. The next section discusses this and shows how

```
1 template<typename E,
2            typename Dummy =
3                typename RpcException< ConcExcp<E, void>, E_TYPE>::d_void >
4 class ConcExcp : public E, public RpcException< ConcExcp<E, Dummy> >
        ⋮
```

LISTING 4.6: *Forcing instantiation of `RpcException` in `ConcExcp`*

programmer-provided inheritance information can be used to allow automatic polymorphic exception mapping.

4.6.4 Polymorphic mapping of thrown exceptions

To be able to navigate through a part of the exception class hierarchy, the base class of each exception class must be known to the method invoker. Since C++ has multiple inheritance (and it is even potentially useful with exception classes, enabling an exception to exist in several independent exception hierarchies), this means a list of base classes. If it can be assumed that RPC counterparts are declared for each non-RPC exception class from a certain inheritance level upwards, it is enough for the programmer to list just immediate base classes. Base classes further up in the inheritance hierarchy can then be found using their base classes.

To provide a list of base classes the programmer must be able to write a C++ expression containing an arbitrary number of types. In current C++ one way to do this is to write a function type declaration, which may contain an arbitrary number of parameters (see Section 2.3.4).[2] The `MapBases` type definition is used to also provide information about the base classes of the non-RPC exception class. The typedef is actually a definition of a function type, where the return type is the non-RPC counterpart of the RPC exception being defined, and the parameter list contains a list of the base classes of the non-RPC exception. An example is shown in Listing 4.7.

When metafunctions inside `RpcException` register exception mappings, they also create a tree of provided inheritance relations, starting from `std::exception`. Later, when a non-RPC exception tries to escape from an active object, the method invoker

[2] The new C++11 contains *variadic templates* which can take an arbitrary number of template parameters. This new language feature will allow a more convenient syntax for base class lists.

```
1 class E1L : public B1L, public B2L { /* ... */ };
2
3 class E1Rpc : public E1L, public RpcException<E1Rpc>
4 {
5 public:
6    typedef E1L MapBases(B1L, B2L); // List also bases of E1L
7    E1Rpc(const E1L& e);
          ⋮
8 };
```

LISTING 4.7: *Providing polymorphic mapping information*

first tries to find an exact mapping using the **typeid** operator, as described earlier. If no exact mapping is found, it starts scanning the inheritance tree. For each node in the tree, the code checks whether the thrown exception can be cast to the type contained in that node (**dynamic_cast** is used for the test). If the test succeeds, the code proceeds down the tree to test the subclasses of the node. This is continued until a leaf node is found. This node then represents the most derived class in the provided hierarchy which is a base class of the thrown exception. An RPC counterpart of the found class is then created and propagated out from the active object.

If derived class exceptions are mapped to their base class RPC counterparts, exception objects become sliced and lose their derived class identities (just as slicing happens in C++ with base class value parameters and return values). This is unavoidable since RPC exception mapping requires the programmer to explicitly declare all possible RPC exception classes, and if derived exceptions are unknown to the programmer, suitable RPC counterparts do not exist. However, since in those cases exceptions are caught using their base classes anyway, slicing does not affect choosing the right exception handler. Of course, slicing can affect information stored inside the exception object itself as well the behaviour of its virtual member functions.

The only way to avoid the slicing problem would be to synthesise the RPC counterpart exception classes during runtime when new exception types are encountered. Creating new classes on the fly is not possible in C++ (nor in most other compiled languages), so the compromise is inevitable.

4.6.5 Providing inheritance information through trait classes

When the `ConcExcp` template is used to declare RPC exceptions with polymorphic mapping, things get more complicated. The `MapBases` definition has to be given to `ConcExcp` somehow, because it can no longer provide it automatically (it does not know the necessary base classes). One solution would be to pass inheritance information as an additional template parameter. However, adding a new template parameter each time more information is needed becomes soon impractical. Therefore *trait* classes are used to provide necessary information.

Traits are a mechanism that is used in many libraries, including the C++ standard library. The C++ standard defines a trait as "a class that encapsulates a set of types and functions necessary for class templates and function templates to manipulate

objects of types for which they are instantiated." [ISO/IEC, 2003, §17.1.18] Traits are useful for providing extra information and functionality for objects which are either not class instances (primitive types, for example), or which come from a third party and as such cannot be modified to include extra information. In the C++ library, "character traits" are used by the `basic_string` class template to obtain information about the character type it operates on [ISO/IEC, 2003, §21.1].

This thesis introduces *concurrency traits* to provide information about concurrent exceptions to `ConcExcp`. A concurrency trait is a struct which defines the `MapBases` typedef (concurrency traits are also used for exception folding information described in Section 5.7). The `ConcExcp` template uses the trait to find out necessary exception mapping information. The trait class provided by the programmer can be a specialisation of class template `ConcTraits`, in which case `ConcExcp` uses it automatically, or it can be a normal struct, in which case it has to be explicitly given to `ConcExcp` as a template parameter(this is also how character traits are used in the C++ standard library). An example of concurrency traits is shown in Listing 4.8.

With concurrency traits it is possible to achieve mapping from third party non-RPC exceptions hierarchies to their RPC counterparts without modifying original exception classes. A definition of a concurrency trait and a `ConcExcp` typedef is

```
1  class E1L : public B1L, public B2L
2  {
3    /* ... */
4  };
5
6  // Concurrency trait as a specialisation of ConcTraits
7  template<>
8  struct ConcTraits<E1L>
9  {
10   typedef E1L MapBases(B1L, B2L); // Mapping information
11 };
12
13 typedef ConcExcp<E1L> E1Rpc; // Uses ConcTraits<E1L>
14
15 // A concurrency trait as a separate struct
16 struct MyOwnB1LTrait
17 {
18   typedef B1L MapBases();
19 };
20
21 typedef ConcExcp<B1L, MyOwnB1LTrait> B1Rpc;
```

LISTING 4.8: *Example of using concurrency traits*

enough for automatic mapping. If the third party library internally throws exceptions which are further down in the inheritance hierarchy than their API reveals, those exceptions are mapped to the closest declared RPC counterpart.

If no concurrency trait is provided for an exception class (as a template specialisation or an explicit struct), the primary definition of the `ConcTraits` template is instantiated. It provides a typedef `MapBases` for exact exception mapping without inheritance information needed for polymorphic mapping.

If the programmer has access to the original non-RPC exception definitions, an external concurrency trait makes the program less readable. For this reason the primary definition of the `ConcTraits` contains a metafunction which looks inside the non-RPC exception class definition. If `MapBases` is found there, it is used to provide polymorphic mapping.

4.7 Implementation issues in RPC exception passing

This section discusses implementation details concerning the exception class registry needed for the re-creation of exception objects on the client side. It addresses the issues of mapping between types and type id strings, as well as concurrency. It also discusses how exceptions are handled in the active object method invoker.

4.7.1 Providing unique IDs for RPC exception classes

The exception registry is used to re-create exception objects from marshalled data, which requires each exception class to be associated with a unique type id that identifies the type of the marshalled exception object. In this respect, the situation is identical to other serialisation mechanisms. C++ and its RTTI provide a method for creating a string representing the "name" of a type, namely `typeid(Type).name()`. Unfortunately, the C++ standard defines this string as "an implementation-defined NTBS" (Null-Terminated Byte String) [ISO/IEC, 2003, §18.5.1]. The standard itself does not guarantee that this string is unique or even that it stays the same if the code is compiled again.

In practise, the situation is much better than what the standard requires. In most current C++ compilers, the type id strings are unique. This is easy for the compiler to implement, since it has to produce unique type ids for the linker anyway. In some compilers, uniqueness is not guaranteed for nameless classes or local classes

(classes with no external linkage), but such classes are hardly useful as exception classes.

C++ ABIs (Application Binary Interface) are designed to promote compiler interoperability, and the format of the type id string is included in many ABIs. For example, the Itanium C++ ABI [CodeSourcery, 2001] used by several compilers specifies that type mangling used in object files is also used for type id strings. Among compilers using the same ABI the type id strings stay constant between files compiled with different compilers.

In some cases it would be best not to rely on compiler-generated type ids. This is obviously the case if the compiler implementation does not guarantee unique type id strings. In addition to this, if the exception mechanism in this chapter is used together with a serialisation library using user-provided type ids, having user-defined exception ids would be consistent.[3] For these reasons the base template `RpcException` accepts a user-defined type id string as an optional parameter. If given, the registry uses the parameter to identify the exception class. If the parameter is omitted, compiler-generated C++ type id string is used. It would be possible to use compile-time selection to make the type id parameter optional only in compilers that are known to provide unique type id strings.

An example of a user-defined type id is shown below (the value of `MyExcp_typeid` would be defined in the same place where the code of `MyExcp` is located):

```
extern const char MyExcp_typeid[];
class MyExcp : public RpcException<MyExcp, MyExcp_typeid> // ...
```

4.7.2 Concurrency issues

In a concurrent environment, synchronisation and mutual exclusion issues must be analysed. The registry is a static member of the exception base class, so it is generated automatically when the program starts. Similarly, all derived exception classes register themselves to the registry before the main routine begins, using the registrator mechanism described earlier in this chapter. After initialisation the registry remains constant, so in a distributed environment registries remain in a consistent state.

[3] Serialisation libraries aimed for persistence usually require user-provided type ids, because type ids should remain the same between different versions of the program. The same can also be true in distributed systems where subsystems run different versions of the software.

If processes are started as separate programs, each program has its own registry. If processes are created later using *fork* or similar mechanisms, all processes automatically get a copy of the already initialised registry. Even a shared-memory environment would not have synchronisation problems though the mechanism described here is not intended for such an environment. The contents of the registry remain static during program execution, so sharing the data structures would not cause problems (depending of course on the thread safety properties of the compiler and its libraries).

Some C++ environments allow dynamic linking of libraries into an already running program. In such environments, the registry mechanism could be enhanced to allow the client to dynamically link in new exception classes, when they are first encountered. However, in a shared-memory environment this would require additional mutual exclusion to atomically update the registry.

4.8 Applicability to other languages

The RPC exception passing mechanism presented in this chapter is needed because the C++ language standard provides no support for concurrency or inter-process communication (and even the new C++11 standard is limited to threads with shared memory). Many other modern programming languages do provide support for concurrency and RPC (e.g., Java) and they also require exception classes to be serialisable. Those languages have no use for the RPC mechanisms of this chapter.

The mechanisms are also heavily based on C++ template metaprogramming, and as metaprogramming facilities on other programming languages differ, applicability to languages with no built-in RPC exception support is limited. In a language with powerful enough run-time reflection, it could be used to map non-RPC exceptions to their RPC counterparts (and possibly generate those counterparts on-the-fly, if the reflection capabilities of the language are powerful enough).

However, applicability of the mechanism is also of interest to existing RPC implementations built on C++, some of which have limited support for exception propagation or no support at all. C++-based RPC implementations must include serialisation support for parameter and return value passing. Since serialisation of primitive types and arbitrary user-defined types (structs and classes) is important, serialisation typically does not require that parameter and return types are derived from a common base class, but use external marshalling and unmarshalling functions.

Adapting the exception serialisation support of this chapter to such RPC implementations should be fairly straightforward. Exceptions only affect the return from an RPC call, so the code for calling an RPC function can be unchanged. However, code calling the actual C++ function on the other end must be changed to catch exceptions. Exceptions derived from the RPC exception base class can be used as such, and exception mapping techniques described in Section 4.6 can be used for other exceptions derived from `std::exception` or other suitable common base class.

The return message of the RPC call must be changed to include a flag indicating whether the call finished successfully or in an exception. In case of an exception, the normal return value data is replaced by the serialised data of the exception, including information on its type. This data can then be used on the caller side to recreate the exception object and throw it, or embed it in a future.

Serialisation is useful in areas outside RPC as well, for example for saving parts of the program state in a file or database. Theoretically, the mechanism presented in this chapter could be used in such contexts, too. However, it is questionable whether such stored program state could contain exceptions. The polymorphic factory based on templates and CRTP inheritance could be of use for generic serialisation in cases where polymorphic serialisation of third party objects is needed in C++. Limitations to this arise from the fact that actual objects to be serialised must be instances of the CRTP-inherited class template rather than the original classes.

In summary, the RPC exception passing mechanism in this chapter is suitable for use in other C++-based RPC implementations, and in a more limited sense in other situations which need polymorphic serialisation in C++.

4.9 Summary

In chapter it has been shown that exception propagation between processes without shared memory, including serialisation and dynamic creation of exception objects, can be implemented as a template-based library. A mechanism for mapping non-RPC enabled exceptions to their RPC counterparts is also presented. The solution requires minimal additional application code and allows the use of existing exception hierarchies.

The presented solution is light-weight and implemented completely using C++ and its template metaprogramming mechanisms. Necessary object factories and

virtual functions are generated automatically. This means that no pre-processors, code generators or separate IDL specifications are needed.

A proof-of-concept implementation of the mechanisms presented in this chapter is implemented as a part of the KC++ library. The source code can be found in http://www.cs.tut.fi/ohj/kcpp/kcpplib-html/.

Chapter 5

Handling concurrent exceptions

Exception handling is a well-established mechanism in sequential programming. Concurrency and asynchronous calls introduce a possibility for multiple concurrent exceptions. This complicates exception handling, especially in languages whose support for exceptions has not originally been designed for concurrency. Futures are a mechanism for handling return values in asynchronous calls. They are affected by concurrent exception handling as well, since exceptions and return values are mutually exclusive in functions. This chapter discusses these problems and presents a concurrent exception handling mechanism for future-based asynchronous C++ programs.

Some ideas in this chapter have been published in [Rintala, 2006], but have been since extended and developed further. The mechanisms presented in this chapter rely on futures as an asynchronous communication mechanism, but are otherwise independent of concurrency mechanisms. Similarly, the mechanisms do not depend on shared or separate address spaces (and so are independent of the serialisation mechanisms presented in Chapter 4).

5.1 Introduction

Exception handling has become common as a means of handling abnormal situations, and most commonly used programming languages now support exceptions. However, basic ideas behind exception handling are far older [Goodenough, 1975]. Exceptions are now considered as a standard way of signalling about exceptional

situations and they have widely replaced the use of specially coded return values, additional boolean state flags, etc.

At the same time, concurrency has become increasingly important in programming. Reasons for this trend include the need for more processing power, asynchronous calls in distributed systems, as well as improving program structure in reactive systems.

Basically, exception handling is about separating exception handling code from "normal" code. This improves readability by structuring the code logically. Exception handling also introduces its own control flow to the program, so that code does not have to explicitly check for every abnormal situation and divert program execution to an appropriate handler.

Because exception handling is also about control flow, concurrency cannot be added to a programming language without affecting exceptions. Concurrency introduces several (usually independent) threads of execution, each of which has its own control flow. This causes several problems to exception handling, some of which are analysed in [Buhr and Mok, 2000]. Asynchronous calls and futures complicate the problem even further. Combination of exception handling and concurrency have also been discussed in [Romanovsky, 2000, Keen and Olsson, 2002, Xu *et al.*, 2000].

There are several ways to combine exception handling and concurrency in an object-oriented programming language, all of which have their benefits and drawbacks [Romanovsky and Kienzle, 2001]. However, when adding concurrency to an originally sequential programming language like C++, the exception handling mechanism has already been fixed, and added concurrency features should be designed to be compatible with the existing mechanisms. This unavoidably means that some compromises have to be made.

5.2 Issues caused by asynchrony and exceptions

Introducing asynchronous calls (and thus concurrency) also affects exception handling, and this has to be taken into account when designing exception handling mechanisms. This section describes issues caused by asynchrony.

If exceptions are propagated from an asynchronous call back to the caller, it is not self-evident where those exceptions should be handled. The caller has already continued after calling the function and may in fact have terminated its execution. Exception handling mechanisms in most languages bind exception handlers to spe-

cific parts of the code (try blocks etc.), and the caller may have left these by the time the exception is raised.

5.2.1 General exception handling issues

Even without concurrency, most programming languages have to react to multiple exceptions in certain situations. In the simplest case, another exception can be thrown in an exception handler. Most languages allow this "stacking" of exceptions, as long as exceptions thrown in the handler are handled to completion during the execution of the same handler. If an exception thrown in a handler escapes the handler, it causes the original exception to be discarded in most systems (C++, Java, and Ada, for example). As a result, in these situations the two exceptions do not compete with each other.

Some languages allow code to be executed after an exception has been thrown, but before it has been caught in a handler. In C++ this is possible with destructors of local objects, in Java with "finally" blocks. If this code throws an exception and does not handle it locally, two competing exceptions exist. C++ reacts to the problem by terminating the execution of the program. Java in turn discards the original exception. This situation is quite similar to a case where two concurrently thrown exceptions occur.

Finally, many systems allow asynchronous signals which can be raised at any time. These signals are usually associated with signal handlers, whose execution interrupts the normal execution of the program. It is possible that another signal is raised while a signal handler is being executed. In POSIX signals [Stevens, 1992, Ch. 10], this is handled by associating each signal handler with a signal mask describing the signals that are allowed to interrupt the handler. The signal masks provide a simple priority scheme for signals. However, even such a scheme does not easily allow handling based on the occurrence of more than one signal.

5.2.2 Exceptions and asynchrony

Asynchronous calls allow the calling thread to continue its execution before the return value of the call is available. Mechanisms like futures make it possible to refer to an asynchronous return value "in advance", but exceptions thrown from an asynchronous call are more problematic.

In C++, exception handlers (catch clauses) are only "active" while the thread of execution is in the respective try block. Therefore, catch clauses can only catch exceptions from asynchronous calls, if the calling thread waits for the call to complete *before* leaving the try block. It should be noted that leaving a try block may be caused by normal program execution or by another thrown exception.

Several asynchronously executing threads also introduce the possibility of several exceptions being raised concurrently. If exceptions are sent to other threads (using mechanisms described in Chapter 4 or similar), this can result in more than one exception being propagated into a *single* thread.

The way return values from asynchronous calls are handled also affects exception handling. In a sense, an exception is an alternative to a return value. In synchronous calls, return values and exceptions are mutually exclusive. If an exception is thrown from a call, a return value is not created, and vice versa. However, mechanisms like futures act as a placeholder for the return value, and they are created *before* the call completes. This means that the placeholder exists and can be accessed even if the call terminates with an exception.

Futures can usually be copied or even sent to other threads before the call completes. This possibility has to be taken into account, if an exception is thrown from an asynchronous call. Exceptions should work consistently in this situation as well. The situation becomes especially interesting if there is more than one thread waiting for a future. Normal sequential exceptions are not usually thrown more than once (unless they are explicitly re-thrown), but propagating an exception to several threads would lead to throwing a copy of the same exception multiple times, once in each thread.

5.2.3 Special issues in C++ exception handling

Exception handling features in the C++ language resemble exceptions in the Ada language, after which parts of the C++ exception handling were modelled. Likewise, Java took most of its exception handling features from C++. The roots of exception handling are of course much deeper (an overview of its history can be found in [Ryder and Soffa, 2003]). However, there are some unique features and limitations in C++ exception handling that have to be taken into account when concurrency is introduced to the language and which limit available options for concurrent exception handling.

In C++ standard, the lifetime of an exception is divided into three parts during its handling [ISO/IEC, 2003, §15.1/7]: An exception is *thrown* when a throw statement is executed. The actual thrown exception object is a *copy* of the object used in the throw statement. An exception becomes *handled* when an appropriate catch clause is found and entered. Finally, an exception is *finished* when the execution exits the catch clause, causing the exception object to be destroyed. This chapter uses terms *thrown exception* and *handled exception* as defined by the C++ standard.

One distinguishing feature in C++ is the fact that local objects must have their destructors executed while searching for the appropriate exception handler. This language feature is called *stack unwinding*. It complicates issues because user code is executed while searching for the exception handler. However, the destruction mechanism allows additional handling code to be executed after an exception is thrown, but before it is handled, or before the execution of a program otherwise leaves a try block.

Because C++ destructors make it possible to execute user code while searching for an exception handler, destructors can throw additional exceptions (a similar situation occurs if copying the exception object throws an exception). Therefore, the C++ language has to consider several thrown exceptions even without concurrency.[1]

Standard C++ allows several thrown exceptions to exist simultaneously as long as they are on different levels and do not compete for the same handlers. When stack unwinding causes a destructor of a local object to be executed, the destructor may throw additional exceptions. C++ requires that these exceptions must be handled in the destructor and may not leak out of it [ISO/IEC, 2003, §15.2/3]. The C++ standard dictates that violating this rule is considered a fatal error and such a program is terminated [ISO/IEC, 2003, §15.5.1].

Thrown C++ exceptions can be nested or "stacked" on top of each other during stack unwinding, but the latest exception must always be handled completely before earlier exception handling continues. When concurrency is introduced, multiple exceptions on the same level have to be handled in a special way because the language itself makes it hard to throw them normally.

In many ways exceptions and return values can be regarded as similar methods for returning from a function. However, one feature that C++ inherits from the C

[1] A similar thing can happen also in Java. When an exception is thrown in a try block, an associated finally-block is always executed. If a new exception is thrown from the finally-block, the original exception is *discarded* and the new exception replaces it [Gosling *et al.*, 2005, §14.20.2]. This demonstrates how difficult it is to cope with multiple concurrent exceptions.

language is that the caller of a function may ignore and discard the return value. In this respect, exceptions are different from return values since they cannot be silently ignored. This distinction becomes important with asynchronous calls and futures, because if a future is a placeholder for both the return value and the possible exception, it also becomes possible to ignore exceptions.

In many languages all exception classes have to be derived from a common base class representing all exceptions. C++ makes no such requirements although it supports inheritance hierarchies in exception handling. However, the lack of a common base class makes it difficult to handle exceptions in a uniform way, since in C++, a common base class is the only way to treat objects polymorphically at run-time. For this reason the mechanism presented in this chapter requires that all concurrent exceptions are derived from a common exception base class as described in Chapter 4. This base class also provides necessary code for serialisation. Unless stated otherwise, all exception classes in this chapter are derived from the common base class (sometimes these classes are called *concurrent* exception classes for emphasis). A mechanism to map normal exceptions to suitable concurrent counterparts was described in Section 4.5.

5.3 Asynchronous calls, futures, and exceptions

If an exception is thrown during the execution of an asynchronous call and is not handled locally, it must be propagated to the caller. Since the call is asynchronous, the only way for an exception to propagate further is through futures. A single asynchronous call may trigger several futures through future reference parameters and futures source. In a concurrent environment there may also be several waiting threads, if futures have been copied to other threads.

When an exception is thrown from an asynchronous call, the library code responsible for handling the call catches the exception. It serialises the exception object. Then it and embeds the data in the reply message of the call, which is sent back to the caller using mechanisms described in Chapter 4.

In the caller, a copy of the exception object is created from the message data and a pointer to this exception object is stored inside the future. This exception object, which is not yet thrown, is called a *pending exception* in this chapter. If the future is copied to another thread, the serialisation code also serialises the pending exception object. If a future is still empty when it is copied, the library code keeps track of

the future. Later, when the future receives its value or an exception, the result is automatically propagated to all copied futures.

Whenever the value of a future is needed, the future checks whether it contains an exception instead of a value. If this is the case, the future throws the pending exception, which is then handled using normal C++ exception handling mechanisms. If the value of the future is accessed another time, the future throws the same exception again. It should be noted that copying the future object itself, assigning it to other futures or passing it as a parameter does not throw the exception. The exception is thrown only when the value of the future is accessed.

Futures throw exceptions they contain only when their value is accessed. This means that if a future is destroyed without accessing its value (or before the value has been received), the only logical choice is to silently destroy the future, even if contained (or will contain) an exception. This follows value semantics used by futures, but is different from the semantics of normal C++ exceptions (which are handled immediately and cannot be implicitly ignored). Future groups discussed in Section 5.4.3 can be used to force synchronisation with a future before it is destroyed, in which case its exceptions can be handled or propagated further.

Exceptions thrown in asynchronous calls must be propagated to *all* affected futures, which implies copying the exception object. This is not in contradiction to normal C++ since the language gives compilers the right to copy exception objects when necessary [ISO/IEC, 2003, §15.1/3]. However, it means that a copy of the same exception may be thrown in several places and may be handled several times in multiple exception handlers when futures are copied and passed to other execution threads. Although this behaviour is logical and practically the only option, programmers must consider its implications in the program logic.

Exceptions should also affect the semantics of future sources (Section 3.3). A future source does not represent a return point from a call. However, if a valid value cannot be bound to a future source, it must be possible to bind an exception in place of the value. The bound exception object is then sent to all generated futures. For this reason future sources provide a way to send pending exception objects to generated futures (without first throwing the exception).

Future sources get their value using the `bind` method. In addition to this, future sources contain a method called `bindThrow`, which takes an exception object as its parameter. The method copies and stores the exception object inside the future source and sends it to generated futures. The same applies also to all futures

that are generated from the future source after `bindThrow` has been called. This manual exception propagation is consistent with the manual value binding, making it straightforward to use.

Since each future source can only contain one value or exception, an attempt to bind the same future source twice results in an exception (only thrown to the binder). Similarly it is an error to destroy a future source without binding it. If this happens, a predefined exception `FutureSrcNotBound` is sent to all generated futures, preventing deadlocks.

5.4 Handling multiple concurrent exceptions

Asynchronous calls make it possible to end up in a situation where several exceptions are raised concurrently. If the caller continues its execution while an asynchronous call is active, both the client and server threads may end up throwing an exception, and both of these exceptions should be propagated in the client thread. The C++ language cannot handle more than one thrown exception on the same level, and this exception cannot be changed before it is caught. This forces some compromises to be made.

5.4.1 Problems caused by multiple exceptions

The C++ exception handling model makes it impossible to resume the execution of a try block after an exception is handled (reasons for not using an alternative resumption model in C++ can be found in [Stroustrup, 1994]). When the C++ exception handling mechanism searches for a correct exception handler, it permanently exits from try blocks, destroying their local variables. This includes the try block which contains the chosen handler. After the exception is finished, program execution continues from the point after the try-catch-compound containing the chosen exception handler.

If several concurrent exceptions end up in one thread, only one of them can be handled at a time. However, handling the first exception means leaving try blocks (and their respective catch-handlers). This would make it impossible to search for handlers for the rest of the exceptions, since all handlers are no longer available (the program execution has already left try blocks). This kind of behaviour would be necessary in situations where there are several independent exceptions, and the caller

wants to react accordingly to more than one of these before letting the exception handling proceed further.

Another problem with handling concurrent exceptions one at a time would be to choose the order in which the exceptions should be handled. Exceptions may be thrown from several (mutually independent) locations, so deciding the correct handling order would either require a global priority scheme for exceptions, or mean that there would have to be a mechanism for informing the relative priority of a thrown exception (and maybe change it during exception handling when the exception is propagated to other parts of the program). Unfortunately, this kind of simple priority scheme is not enough for all programs. It would require that all sources of exceptions are aware of all possible exceptions so that priority ordering can be defined.

The third and maybe the most important aspect in concurrent exception handling is the fact that several concurrently thrown exceptions may in fact be caused by the same abnormal situation. If thrown exception objects have the same cause and contain the same data, they could be reduced to a single exception. However, sometimes the nature of the actual abnormal situation may only be understood by analysing *all* of the exceptions it causes, in which case it is important that the exception handling mechanism can cope with several pending exceptions. In certain cases it would also be beneficial to be able to *replace* a set of exceptions (caused by the same abnormal situation) with a new exception representing the whole exceptional situation. For example, several timeout exceptions from processes running on a remote machine could be mapped to an exception representing connection failure to the whole machine.

Writing exception handling becomes easier if these exception reductions can be performed *before* an exception handler is chosen. An exception handler can then catch a single exception whose type represents the whole situation. It is impossible to give a global rule for reducing multiple exceptions to one, since reductions depend on the context where exceptions occur. It is important that the program is allowed to provide its own algorithms for reduction.

Figure 5.1 on the following page shows the structure of the concurrent exception handling mechanism. It is based on futures and future sources, as well as *future groups* for collective synchronisation, *compound exceptions* for handling multiple exceptions, and *reduction contexts* and *reduction functions* for exception analysis and reduction. These mechanisms have been influenced by earlier works on

exception reduction, resolution and concertation, for example [Romanovsky, 2000, Xu *et al.*, 2000, Issarny, 2001]. The behaviour of exception reduction is described in the following sections.

5.4.2 Compound exceptions

The C++ language can only propagate one exception at a time. Another exception may be raised during stack unwinding triggered by the first exception, but these additional exceptions must be handled to conclusion before stack unwinding proceeds further. In a concurrent program this limitation is problematic, because several exceptions may need to be propagated from asynchronous calls to a single try block.

A *compound exception* class represents a set of exceptions. It is a normal concurrent exception class, but its instances can contain an unlimited number of other

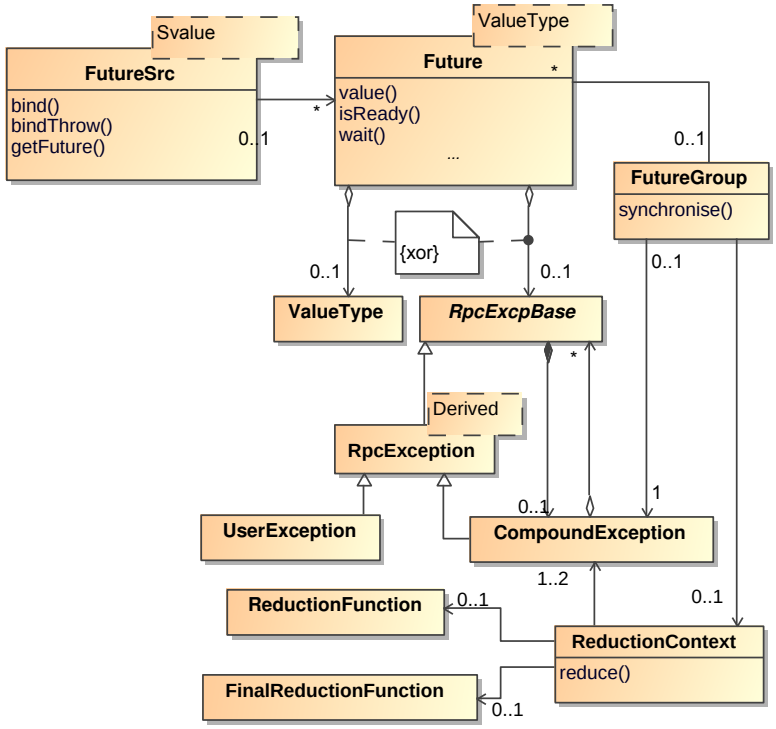

FIGURE 5.1: *Structure of concurrent exception handling*

exceptions. As compound exceptions are concurrent exceptions, they can be passed among execution threads. Usually, compound exceptions are created by future groups, which collect exceptions from several futures to a compound exception. Compound exceptions are then used by reduction contexts and passed to reduction functions for analysis and manipulation.

The interface of the compound exception class allows addition of new exceptions and removal of existing ones. In addition, all exceptions may be moved from one compound exception to another, making it possible to combine compound exceptions in reduction functions or exception handlers. Compound exceptions also provide iterators to iterate through all exceptions of a certain type.

During exception reduction, it is useful to replace a compound exception with a selected top-priority exception. However, it is still be useful to also store the original "secondary" exceptions, in case they are useful later. In this thesis, *all* concurrent exception objects may contain an optional compound exception object where secondary exceptions may be stored. This approach is somewhat similar to chained exceptions in Java, where each exception may contain a reference to its *cause*, which is another exception [Sun Microsystems, Inc., 2006, Throwable].

Internal C++ exception handling code is allowed to copy exception objects without restriction. However, compound exception objects may be quite large since they consist of several other exceptions. For this reason copies of compound exceptions share their contents using reference counting and copy-on-write (COW) semantics. The original compound exception and its copies share their exception objects until a change is made to any of them, in which case a real copy of the contained exception object is made.

Usually exceptions originate from futures, and their lifetime may be shorter or longer than the lifetime of the compound exception their exceptions are added to. Futures can be destroyed as a part of stack unwinding during exception handling, or their existence can continue after exception handling has been completed. For these reasons reference-counted sharing is used between futures and compound exceptions, too. They both share the same exception object and use reference counting to find out when it is safe to destroy the object. Copy-on-write semantics is not used here, since the intention is to really share the same exception object, not just avoid duplication overhead. Sharing the exception means that even after exception handling has been completed using the compound exception, the futures (if they still exist) throw the same exception object again, if their value is accessed.

5.4.3 Future groups

Sometimes "delayed" exceptions caused by asynchronous calls and futures are what the programmer wants, but in many cases a try block needs to contain multiple asynchronous calls, and the programmer wants to know if any of them ends with an exception before leaving the try block. Checking each future separately would be awkward and not even straightforward in the case of multiple exceptions. For these reasons *future groups* are provided to help with synchronised exception handling.

Future groups are objects to which futures may be registered. They have an operation `synchronise`, which waits for all the futures in the group to receive a value (or a pending exception). In this respect future groups are similar to barrier synchronisation [Andrews, 1991, Ch. 4] and future sets in ES-Kit [Chatterjee, 1989]. If pending exceptions are found in the futures during synchronisation, the future group collects these exceptions in a compound exception. The exceptions in the compound exception are later reduced (Section 5.5), and an exception suitable for the situation is thrown. If a reduction context is registered with the future group, the synchronisation of the future group automatically triggers reduction.

When the program asks for a value of a future belonging to a future group, the future first waits for its asynchronous call to complete. Then, if the call ended with an exception, the future asks its future group to perform synchronisation. This makes sure that all exceptions in the group are available before exception handling is started. Finally, if possible, exception reduction is performed, and an appropriate exception is thrown. If no reduction information is given, a compound exception containing all the exceptions is thrown. The code in Listing 5.1 on the next page demonstrates the use of a future group in a try block. The future group in the code is not given reduction information, so it throws a compound exception if exceptions are found.

If a future receives a normal return value instead of an exception, it directly returns this value without synchronising the future group, since no exception reduction is needed yet. An alternative strategy would be to perform synchronisation also in this case. However, always forcing synchronisation would seriously limit the amount of asynchrony in the program. However, a future group can be asked to perform explicit synchronisation at any point, if needed.

If a future group is destroyed before its synchronisation is called, its destructor performs synchronisation automatically. This makes future groups very close to the Resource Acquisition Is Initialisation (RAII) idiom [Stroustrup, 2000, §14.4], which

```
1  Future<int> f1;
2  try
3  {
4     FutureGroup group;
5     f1 = call1(); // An asynchronous call
6     group.add(f1);          // Register f1 to the group
7     group.add(call2());          // Register without storing the future
8     int i = f1.value(); // Synchronises with group first
9  } // Destruction of group synchronises with all futures
10 catch (CompoundException const& e1)
          ⋮
```

LISTING 5.1: *Using future groups to synchronise several futures*

is very common in C++ for exception-safe resource management. In RAII each resource is wrapped inside an object whose destructor releases the resource, preventing the possibility of resource leaks even in case of exceptions. Future groups represent the responsibility to synchronise with one or more futures (asynchronous calls), and their destructor makes sure this responsibility is fulfilled.

5.5 Exception reduction

As mentioned previously, several futures in a try block may receive pending exceptions. In that case it would be beneficial if these exceptions were reduced to a single exception object representing the complete situation. Even if the exceptions are not the result of the same cause, finding "the most important" of the exceptions often depends on program context.

Because there is no single all-purpose way to reduce a set of exceptions to a single exception or a smaller set of exceptions, programs should not be forced to any predefined behaviour. In the mechanism described in this thesis, *reduction contexts* are objects which manage exception reduction strategies in different parts of the program. Reduction contexts allow programs to bind *reduction functions* to them. Reduction functions analyse the current set of pending exceptions, alter it, and select an appropriate exception to be thrown.

5.5.1 Reduction contexts

Future groups are responsible for grouping exceptions received through futures belonging to the group. Reduction contexts are objects which are responsible for stor-

ing the received exceptions and performing exception reduction on them. They also contain information about functions needed for reduction.

Reduction contexts are introduced because of separation of concerns. Future groups should only be responsible for synchronising with a group of futures and collecting their exceptions. This allows future groups to be created as local variables in a try-block, in which case they are destroyed automatically at the end of the block and perform synchronisation before the try-block is exited.

A reduction context contains a compound exception, one or two reduction functions, and an optional exception store (another compound exception). The interface of the class contains methods to perform reduction and to check whether reduction has already been completed.

Reduction contexts are designed to be used in the following way: First the compound exception of the context is filled with appropriate exceptions (by future groups). Then one of the reduction methods of the context is called (by a future group or manually). This in turn calls a reduction function, which analyses the exceptions and selects a suitable exception to represent the situation. The reduction functions may also insert, change, and remove the stored exceptions. Finally, the reduction method of the context throws or returns the exception produced by reduction.

The reduction context class provides three different methods for performing the reduction, `reduce()`, `reduceNoThrow()`, and `reduceThrown()`. The first method reduces collected exceptions and throws the resulting exception. If throwing the exception is not necessary, `reduceNoThrow()` can be used to return the exception object instead of throwing it. Finally, `reduceThrown()` can be used in situations where an exception has already been thrown, and throwing another exception is not possible (this situation is discussed in Section 5.5.4).

The method `reduction_complete()` is provided for checking whether reduction has been completed. It is allowed that the first reduction only reduces some of the exceptions and leaves the rest for another reduction pass. For this reason, `reduction_complete()` returns true only if the compound exception of the context becomes empty after reduction, or if the last reduction didn't result in an exception (reduction could not find anything to reduce). The reduction status can also be set or reset again, if necessary (for example if new exceptions are added to the context).

There are several ways to connect a reduction context to a future group, depending on how the future group is constructed. In the simplest scenario, the future

group is created using its default constructor. In this case, no reduction is performed and no reduction contexts are created. If an exception occurs, the future group synchronises with its futures and throws the resulting compound exception.

In the second scenario, the future group constructor is given a compound exception. In this case, the future group itself does not trigger reduction, it only collects its exceptions to the compound exception (and throws a copy of it, if necessary). It should be noted that the compound exception may belong to a reduction context (or be added to a reduction context later), in which case reduction can be performed afterwards.

If the future group itself should perform reduction, its constructor can be given a reduction context object. In this case, the future group gathers its exceptions in the reduction context and triggers reduction before throwing an exception.

If an external reduction context is not needed, future groups also have a constructor which accepts reduction functions as its parameters. In this case, the future group creates its own internal reduction context, which uses provided reduction functions to reduce collected exceptions.

The compound exception object inside a reduction context is either created automatically when the reduction context is created or it can be an external object given to the reduction context during construction. In the first case, the compound exception object is destroyed automatically when the reduction context is destroyed. The second case allows the program to keep the compound exception (containing remaining unreduced exceptions) even after the reduction context has been destroyed.

5.5.2 Reduction functions

Reduction functions are user-defined functions (or function objects) that can be added to reduction contexts to perform exception reduction. When a reduction context is asked to perform reduction, it calls its reduction function and passes the compound exception of the context to it.

Normal reduction functions are used when exception reduction is needed and exceptions have not already been thrown. They receive as parameters a pointer to the compound exception object containing the exceptions, and an optional pointer to an external exception store for storing the remaining exception objects.

Reduction functions use these parameters to decide how to simplify or modify a given set of exceptions. They may add and remove exceptions or transfer them from one parameter to another. Every concurrent exception object can contain an additional compound exception object, so exceptions can also be added to and removed from another concurrent exception object, if needed. For example, a reduction function can decide to embed all remaining exceptions from the compound exception of the reduction context as "sub-exceptions" of the exception that is the result of reduction.

After analysis, a reduction function may return an exception which it regards as the "most important" of the exceptions. Alternatively, it may return a completely new exception object representing the most appropriate exceptional situation as well as alter, insert and remove exceptions in the compound exception.

In many programs, choosing one exception object or throwing a compound exception is enough, but sometimes it is useful to collect exceptions from (at least some) future groups into an external exception store to be handled later. For example, an already detected higher priority exception may make a lower level exception unnecessary, but they could still be stored for logging purposes. For this purpose each reduction context can be given an additional compound exception object, which is passed to reduction functions as an external exception store. Reduction functions may use this compound exception as a store for exceptions that are not thrown or embedded in other exceptions.

The C++ language contains a limitation: when an exception has been thrown, it cannot be replaced by another exception until it has been caught in a catch-clause. This means that reduction cannot change the type of a thrown but not handled exception, even if a more important exception is found, or if the thrown exception should be changed to a more appropriate exception. However, since future groups automatically perform synchronisation and reduction, reduction is usually performed *before* exceptions have been thrown.

Thrown-reduction functions are used when reduction is triggered in a situation where a thrown exception has already caused exception handling to begin. Thrown-reduction functions receive the thrown exception object as an extra parameter if the thrown exception is an RPC-exception. If a normal non-RPC exception has been thrown, a null pointer is passed instead (since there is no way to refer to an arbitrary exception object). Since a new exception cannot be thrown if a thrown exception already exists, thrown-reduction functions return nothing.

Future groups call the thrown-reduction function instead of normal reduction if they notice that an exception has already been thrown before reduction. C++ poses some limitations to this reduction, but these are corner cases which do not affect the majority of code. Section 5.8.2 discusses the potential problems.

It is possible that a reduction context is not given a thrown-reduction function, or that the thrown-reduction function does not remove all exceptions from the compound exception to be reduced. In those cases, the remaining exception are left in the reduction context.

Not being able to perform normal reduction because of an already-thrown exception makes reduction more complex and limits the options available for reduction functions. One solution to this problem is to catch the thrown exception normally in a catch clause, insert it into the reduction context and trigger reduction manually after that. Suitable macros can be written for such a purpose.

Since the same reduction strategy may be needed in several reduction contexts, class template `ReductionContextRF` is provided. It takes a pointer to a reduction function and an optional pointer to a thrown-reduction function as template parameters. The template is derived from `ReductionContext` and its constructors register the reduction functions with the context. This way it is easy to write a typedef for a reduction context with suitable reduction functions.

5.5.3 Example of reduction

Listing 5.2 on the following page shows a simple example using a future group and a reduction function. In the example, `Server` (implementation not shown) is responsible for concurrently acquiring and summing up integers. If all necessary numbers cannot be acquired, the server throws a `MissingNumber` exception, which contains a list of ids of missing numbers (the exception is a struct to make the example shorter). Template inheritance used to derive new concurrent exception classes was described in Chapter 4.

The example uses a reduction function `reductionFunc` to reduce exceptions. This simple reduction does not need an external exception store, so it ignores its second parameter. It only uses the first parameter, a compound exception to which exceptions from futures have been collected by the future group. The reduction function uses exception iterators to iterate through all exceptions of type `MissingNumber`. It creates a new exception object (line 13) and copies all missing numbers into it

```
1  struct MissingNumber : public RpcException<MissingNumber>
2  { // A simple exception class
3    vector<string> ids;
4  };
5
6  // Definitions of operators << and >> for MissingNumber removed
         ⋮
7  RpcExcpBase* reductionFunc(CompoundException* groupCE,
8                             CompoundException* /* exception store not needed */)
9  {
10   RpcExcpIterator<MissingNumber> it = groupCE->begin();
11   if (it != groupCE->end())
12   { // MissingNumbers found in compound
13     MissingNumber* mn = new MissingNumber; // Final exception
14     while (it != groupCE->end())
15     { // Iterate, add numbers to mn
16       mn->ids.insert(mn->ids.end(), it->ids.begin(), it->ids.end());
17       it = groupCE->erase(it); // Remove exception, get next
18     }
19     // Insert the rest of exceptions into mn
20     mn->getCE()->insert(groupCE->begin(), groupCE->end());
21     return mn; // Return the list of all missing numbers
22   }
23   } else if (!groupCE->empty())
24   { // Otherwise the whole compound exception is thrown
25     return groupCE;
26   } else { return 0; } // No exceptions at all
27 }
28
29 int combine(Server& s1, Server& s2)
30 {
31   ReductionContext rc(&reductionFunc);
32   try
33   {
34     FutureGroup fg(rc); // Or directly fg(&reductionFunc)
35     Future<int> n1 = s1.sumNums(); fg.add(n1);
36     Future<int> n2 = s2.sumNums(); fg.add(n2);
37     return n1.value()+n2.value();
38   }
39   catch (const MissingNumber& mn)
40   {
41     cerr << "Missing numbers:";
42     for (unsigned int i=0; i<mn.ids.size(); ++i)
43     {
44       cerr << " " << mn.ids[i];
45     }
46     cerr << endl;
47     abort();
48   }
49 }
```

LISTING 5.2: *Example of future groups and reduction functions*

(line 20). This exception object is then returned as the actual exception to throw. If no `MissingNumber` exceptions are found, the original compound exception is returned.

Function `combine` creates a future group, makes two asynchronous calls and adds their futures to the future group (lines 34–36). If either of those calls ends with an exception, the future group synchronises with both futures and calls the reduction function. The reduction function reduces `MissingNumber` exceptions to a single exception, which is then thrown and caught by the catch clause (line 39). In this example, all exceptions except `MissingNumber` are considered secondary, and are ignored by the reduction function unless no missing numbers are found.

The code in Listing 5.3 shows a simplified way to handle multiple exceptions in a loop, using a reduction function to decide on what exceptions to throw. The loop continues to throw new exceptions until reduction has emptied the compound exception or until exception handling forces the program to leave the loop.

5.5.4 Handling already thrown exceptions

When an exception has been thrown, it cannot be replaced by another exception until it has been caught in a catch-clause. This means that reduction functions cannot

```
1  ReductionContext rc(&reductionFunc, &thrownReductionFunc);
2  while (!rc.reduction_complete()) // While reduction is not complete
3    try
4    {
5      if (!rc.reduced())
6      { // not reduced -> first time
7        FutureGroup fg(rc); // Future group using reduction context
8        Future<int> n1 = s1.sumNums(); fg.add(n1);
9        Future<int> n2 = s2.sumNums(); fg.add(n2);
10     } // Automatic synchronisation of future group here
11     rc.reduce(); // Reduce and (possibly) throw
12   }
13   catch (const MissingNumber& e)
14   { // Exception handling
         ⋮
15     continue; // Deal with next exception, if any
16     // or: break; // Continue, abandon the rest of exceptions
17     // or: throw; // Abandon rest of exceptions, rethrow
18   }
```

LISTING 5.3: *Using a compound exception and reduction in a loop*

change the type of a thrown but not handled exception. Future groups perform necessary reduction *before* pending exceptions are actually thrown, but it is of course possible that an exception is thrown from another source than a future.

If an exception originates from outside future groups, the future groups can perform minimal reduction using thrown-reduction functions. They can embed the exceptions of the group to the thrown exception or otherwise handle them. Since the thrown exception cannot be replaced, this kind of reduction is inadequate. For example, the thrown exception may represent a minor problem while the exceptions from the future group are of a more critical type. However, C++ does not allow any control of exception handling before a catch block is entered, therefore exception reduction cannot interfere by adding exception reduction.

One solution to this problem is for the programmer to catch the thrown exception and perform reduction in the catch clause. The caught exception can be added to the reduction context and normal reduction performed. Listing 5.4 contains an example which shows how this can be achieved.

```
1  ReductionContext rc(&reductionFunc); // No thrown reduction
2
3  try
4  {
5     FutureGroup fg(rc); // Future group using ce as compound
6
7     Future<int> n1 = s1.sumNums(); fg.add(n1);
8     Future<int> n2 = s2.sumNums(); fg.add(n2);
9
10    throw MissingNumber(); // Normal throw, reduction not done
11 }
12 catch (const RpcExcpBase& e)
13 {
14    if (!rc.reduced())
15    {
16       // Reduction not performed, a rogue exception
17       rc.addException(e); // Now all exceptions are together
18       rc.reduce(); // Reduce and throw
19    }
20    else
21    {
22       // Exception is the result of reduction, nothing to do
23       throw; // Propagate the exception further
24    }
25 }
```

LISTING 5.4: *Catching and reducing rogue exceptions*

5.5.5 Reduction classes

It would be useful to provide a library of ready-made reduction components, from which the programmer could choose and combine a suitable reduction algorithm (possibly by adding reduction components of her own). Some basic tools for this can be implemented using template metaprogramming. The principles of combining reduction functions will be covered in Section 5.6.

Although using normal functions for exception reduction is a simple way to connect reduction algorithms to reduction contexts, the solution has one limitation. Since functions in C++ are identified by their address, reduction functions look the same to the type system, which prevents most template metaprogramming. The problem can be solved by embedding reduction functions into a class as static members. Then the class itself can be used by template metaprograms to combine reductions. Such classes are called *reduction classes*.

A reduction class is a class (or a struct) with a static member function `reduce` and/or a static member function `thrownreduce`. As the names suggest, the former performs normal reduction while the second is used for thrown-reduction. A reduction class may define one or both of these functions. When reduction classes are combined using template metaprograms, the combining metaprograms check that each class implements the needed reductions. Usually, reduction classes only contain these static members, so the classes are never used to create objects. In theory, C++ namespaces would be much more suitable for this purpose than classes, but namespaces are not types in C++, so they cannot be used for template metaprogramming.

Reduction classes are also a convenient abstraction because they contain both normal reduction and thrown-reduction. Reduction contexts and future groups have appropriate templates to accept reduction classes in addition to pointers to reduction functions. A template `ReductionContextRC` is also provided. This template is similar to `ReductionContextRF` but it takes a reduction class as its parameter, making it possible to write typedefs for reduction contexts using a specified reduction class.

Conversions between reduction classes and reduction functions are not difficult. If `RC` is a reduction class, `&RC::reduce` and `&RC::thrownreduce` produce appropriate reduction function pointers. For the other direction, template `ReducFP<&rf,&trf>` can be provided. It produces a reduction class whose `reduce` calls `rf()` and whose `thrownreduce` calls `trf()`. Similarly, `ReducFP<&rf>` and `ThrownReducFP<&trf>` can be used to get reduction classes with only one of the reductions.

5.6 Combining reduction functions using metaprogramming

In the system described in this thesis, the provided library can take care of some aspects of exception handling, like exception propagation among threads, as well as synchronising exceptions and collecting them to groups. However, the actual implementation of reduction functions must be provided by the application programmer. This is unavoidable, since the logic needed in exception reduction depends greatly on the context where it is performed.

Even if the actual reduction logic varies from situation to situation, there are many tasks which are common to many reduction cases. For example, in many cases it is useful to choose the most important exception from a list where exception types are ordered based on their importance. Similarly, the actual reduction logic could contain a step where the remaining (possibly less critical) exceptions are stored into an external exception store after performing the actual reduction step.

It would be useful if the application programmer could select and combine the reduction logic from ready-made parts and only write those parts of reduction which are less common. To allow this, a framework based on template metaprogramming is described. It allows the creation of reduction classes by combining them from ready-made and user-defined parts.

The mechanism described below is a prototype to show the principles of reduction combinators. It would require further analysis to know which combining strategies and reduction logic steps are so common that they should be implemented as part of the library. However, nothing prevents the application programmer from augmenting the library by writing her own combinator classes and reduction steps.

Since C++ template metaprogramming operates on types, only reduction classes can be combined using the mechanism. If necessary, `ReducFP` template can be used to convert normal reduction functions into reduction classes.

The idea behind combining reductions is to define *combinator classes* which are able to combine several reduction classes to one. Different combinator classes use a different algorithm for doing this. The most important part in combining reduction classes is a template `Reduc<Combinator(RClass1, RClass2, ...)>`. It takes as parameters a combinator class `Combinator` and a list of reduction classes. It produces a reduction class which uses the combinator to select how individual reduc-

tion classes are called and how their results are combined. For example, depending on the combinator used, the resulting reduction class can call the given reduction classes in sequence until one of them chooses and returns an exception.

5.6.1 Combinator classes

Combinator classes are classes which define how reduction classes should be used together to create a combined reduction class. Currently, the following combinator classes are implemented in the KC++ library:

- `Choose` calls the reduction classes in sequence until a reduction returns something. That result is the result of the whole reduction and the rest of reduction classes are not called. `Choose` only makes sense for normal reduction, it does not provide `thrownreduce`.

- `ChooseAndThrown` calls the reduction classes in sequence until a reduction returns something. Then it calls thrown-reduction for the rest of the reduction classes. Calling `thrownreduce` simply calls thrown-reduction of all the reduction classes in sequence.

- `Chain` also calls the reduction classes in sequence. However, it passes the result of the first reduction class to the next reduction class as the *thrown exception*. This way the reduction classes can be used to manipulate the remaining exceptions after the first reduction class chooses the resulting exception. The first reduction class must implement `reduce`, but it can also implement `thrownreduce`. The rest of reduction classes must implement `thrownreduce`.

- `ThrownChain` is just like `Chain`, except that it only provides `thrownreduce`, and `thrownreduce` of the first reduction class is used.

- `Priority` is somewhat different from other combinators. It takes as its parameters a list of exception types instead of reduction classes. Priority produces a reduction which selects an exception based on their priority. It checks whether exceptions of a given types exist and returns the first exception of the first possible listed exception type. Only `reduce` is provided. `Priority` is provided as a shortcut, the same behaviour can also be reached by combining `Choose` and a list of `Picks` (described in the next section).

5.6.2 Ready-made reduction classes

The following basic reduction classes are provided and can be used with combinator classes and user-defined reduction classes. The classes are meant to work as examples, building an inventory of often needed reduction steps would require additional analysis and work.

- `Pick<Exception>`. This reduction class template produces a reduction class that returns the first found exception of a given type. Only `reduce` is implemented, since the operation makes no sense if an exception has already been thrown.

- `Remove<Exception>`. This reduction class template produces a reduction class that remove all exceptions of a given type. Both `reduce` and `thrownreduce` are provided, and `reduce` returns no exceptions.

- `CombineRC<Reduce,ThrownReduce>`. Combines reduction functions of two reduction classes by using normal reduction from `Reduce` and thrown reduction from `ThrownReduce`. A special `CombineRC<Reduce>` with a single parameter is allowed. It produces a reduction class with normal reduction from `Reduce` but no thrown reduction (i.e. `Reduce::thrownreduce` is discarded, if present).

- `ThrowCompound` simply returns the whole compound exception as the result of reduction. This is the default reduction used by reduction contexts, if no explicit reduction is set.

- `ThrowOneOrCompound` is otherwise similar to `ThrowCompound`, except that if there is only one exception to be reduced, that single exception is returned instead of the whole compound exception.

- `ThrownCompoundToStore`, `ThrownExcpsToStore`, `ThrownCompoundToThrown`, `ThrownExcpsToThrown`. These reduction classes are for `thrownreduce` reduction only. They store the exceptions to the exception store or embed them in the already thrown or selected exception. They are meant to be used as later stages with `Chain` or `ThrownChain` in order to store the remaining exceptions to a suitable place. The `ThrownCompound...` reductions store the whole compound exception object, whereas the `ThrownExcps...` versions store the individual exceptions in the compound exception.

- `ThrownRemoveThrown`. This `thrownreduce` reduction simple checks whether the thrown exception also exists in the compound exception to be reduced. If this is true, the thrown exception is removed from the compound exception. For example,

```
Reduc< Chain(Pick<E>,ThrownRemoveThrown,ThrownExcpsToThrown) >
```

  produces a reduction which checks if exceptions of type `E` exist, and chooses and removes the first of them. The remaining exceptions are embedded in the selected exception.

- `ClearCompound` clears all exceptions from the reduction context. This reduction step can be used as the last step to make sure nothing remains in the reduction context.

- `ReplaceCompounds` can be used to "strip" compound exceptions in the reduction context. The reduction step replaces all compound exceptions in the reduction context with their contents, but returns nothing. This step can be used in the beginning of a reduction chain if it is expected that reduction context receives compound exceptions from futures.

- `UniqueIdentity` removes duplicate exceptions from the reduction context. This reduction step can be used if it is possible that the same exception is propagated to the reduction context through several channels.

5.6.3 Example of reduction combining

Listing 5.5 on the next page shows an example of combining reduction classes. The example is the same as in Listing 5.2, but now `FutureSrcNotBound` exceptions participate in reduction and are considered more important than `MissingNumbers`.

The `Choose` combinator is used to choose the first reduction step that returns something. `Pick` returns a `FutureSrcNotBound` exception, if such exceptions exist in the reduction set. If not, the user-provided reduction function is used to reduce `MissingNumber` exceptions. Since reduction combinators work on reduction classes and not functions, `ReducFP` template is used to create a reduction class which calls the user's reduction function. If neither of the reduction steps returned anything, `ThrowCompound` selects the whole reduction set as a compound exception to be thrown.

```
1  // Definition of MissingNumber as before
2  //
3  // Choose primarily FutureSrcNotBound, secondarily the result of reductionFunc,
4  // throw the whole compound exception if neither matches.
5  // Name the resulting reduction class MyReduc
6  typedef Reduc<Choose(
7                      Pick<FutureSrcNotBound>,
8                      ReducFP<&reductionFunc>,
9                      ThrowCompound
10                          )> MyReduc;
11
12 int combine2(Server& s1, Server& s2)
13 {
14    ReductionContextRC<MyReduc> rc;
15    try
16    {
17       FutureGroup fg(rc);
18       Future<int> n1 = s1.sumNums(); fg.add(n1);
19       Future<int> n2 = s2.sumNums(); fg.add(n2);
20       return n1.value()+n2.value();
21    }
22    catch (const MissingNumber& mn)
23    {
24       cerr << "Missing numbers:";
25       for (unsigned int i=0; i<mn.ids.size(); ++i)
26       { cerr << " " << mn.ids[i]; }
27       cerr << endl;
28       abort();
29    }
30    catch (const FutureSrcNotBound& nb)
31    {
32       cerr << "Futures missing their values!" << endl;
33       abort();
34    }
35 }
```

LISTING 5.5: *Combining reduction functions using combinator classes*

5.6.4 Implementation of combinator classes

The parameter `C(R1,R2,...)`in the `Reduc` template uses a function type to provide a list of arbitrary types as described in Section 2.3.4. Here the syntax is used to define a small "domain specific language" (DSL) for combining reduction classes. As variadic templates were added to the new C++11, they could be used instead.

Combinator classes are simply structs with two nested reduction class definitions, one normal class and one template. These are used by the `Reduc` template in a Lisp-like fashion to produce a list consisting of nested pairs. The goal is to achieve a combination of an arbitrary number of reduction classes by defining how two reduc-

tion classes are combined. The reduction classes defined by the combinator class are the following:

- `Nil` is the "terminating" reduction class. It defines the reduction performed when no parameters are present. Thus `Reduc<C()>` performs the same reduction as `C::Nil`.

- `Cons<RC1, RC2>` is a reduction class template taking two reduction classes as its parameter. The first parameter is one of the reduction class parameters of `Reduc`, and the second parameter is the result of combining *the rest* of the parameters. The implementation of `reduce` and `thrownreduce` in `Cons` decides how to call `reduce` and `thrownreduce` in the parameters of `Cons` in order to achieve a suitable combination of reductions.

The `Reduc` template constructs the combined reduction class by recursively instantiating the `C::Cons` template. During instantiation of `Reduc<C(R1,R2,...,Rn)>`, `C:Cons` is instantiated by passing `R1` and `Reduc<C(R2,R3,...,Rn)>` as parameters. As mentioned before, `Reduc<C()>` produces `C::Nil`. This results in a recursive chain of `Cons` instantiations where each `Cons` combines one reduction class parameter with the result of combining the rest of reduction class parameters, terminating with `Nil`.

Below is an example of a stepwise instantiation of `Reduc<C(R1,R2)>`:

1. `Reduc<C(R1,R2)>` ⇒
2. `C::Cons< R1, Reduc<C(R2)> >` ⇒
3. `C::Cons< R1, C::Cons< R2, Reduc<C()> > >` ⇒
4. `C::Cons< R1, C::Cons< R2, C::Nil > >`

(The actual instantiation chain used in KC++ is somewhat different, since typedef `Reduc::Func` is needed to name the calculated reduction class.)

With the `Reduc` framework provided, it is quite straightforward to write application specific reduction combinators. All that has to be done is to write a struct with an appropriate `Nil` reduction class and `Cons` reduction class template. As an example of this, listing 5.6 on the following page shows the implementation of the combinator `Choose`.

The actual implementation of the `Reduc` template is based purely on partial template specialisations, and the primary `Reduc` template is empty. One specialisation

```
1  // Combinator metafunction which calls reduction objects in sequence,
2  // returning the value of first one which doesn't return 0
3  struct Choose
4  {
5     // With nothing to choose, does nothing
6     struct Nil
7     {
8        static RpcExcpBase*
9        reduce(CompoundException*, CompoundException*)
10       {
11          return 0;
12       }
13    };
14
15    // Calls the first param, returns if it didn't return 0
16    // Otherwise calls the second one and returns it value
17    template<typename Head, typename Tail>
18    struct Cons
19    {
20       static RpcExcpBase*
21       reduce(CompoundException* groupCE, CompoundException* storeCE)
22       {
23          RpcExcpBase* headres = Head::reduce(groupCE, storeCE);
24          if (headres) { return headres; }
25          RpcExcpBase* tailres = Tail::reduce(groupCE, storeCE);
26          return tailres;
27       }
28    };
29 };
```

LISTING 5.6: *Implementation of the combinator* `Choose`

is written for each number of parameters (reduction classes). In current KC++ implementation up to eight reduction classes are supported, but it is trivial to increase this limit to any number. Each partial specialisation gets the combinator class and the reduction classes as its parameters, enabling it to pass them to `C::Cons` and a specialisation of `Reduc` with one less parameters. Listing 5.7 on the next page shows the specialisation of `Reduc` with three reduction class parameters.

5.7 Reduction based on inheritance hierarchies

One possible reduction strategy is to find the most derived common base class for all the exceptions, and replace the set of exceptions with a single exception object of that type. This kind of reduction can be convenient as it provides a general way

```
1  template <typename Comb, typename T1, typename T2, typename T3>
2  struct Reduc<Comb(T1, T2, T3)>
3  {
4    typedef typename Comb::template Cons<T1,
5      typename Reduc<Comb(T2, T3)>::Func> Func;
6
7    static RpcExcpBase* reduce(CompoundException* groupCE, CompoundException* storeCE)
8    {
9      return Func::reduce(groupCE, storeCE);
10   }
11
12   static void thrownreduce(CompoundException* groupCE, CompoundException* storeCE,
13                            RpcExcpBase* thrown)
14   {
15     Func::thrownreduce(groupCE, storeCE, thrown);
16   }
17 };
```

LISTING 5.7: *Partial specialisation of* **Reduc** *for three reduction classes*

of reducing an arbitrary set of exceptions. In this thesis, this is called *folding* of exceptions.

For example, if an overflow exception and a division by zero exception have occurred, and if they are both derived from arithmetic exception, they could be reduced to a single arithmetic exception.

On the other hand, this kind of reduction loses information about the types of individual exceptions. It does not allow certain exception types to have a higher precedence than others. If a fatal exception and a minor exception are reduced, the result is their common base class, which abstracts the types of individual exceptions away. Catching this base class exception object does not reveal whether a fatal exception has occurred.

5.7.1 Problems in implementing inheritance based reduction in C++

Even though inheritance hierarchy based reduction is not suitable for all reduction situations, it was considered useful enough to be implemented. There are two problems to be solved. One is related to exception objects and data they contain. The other is to (again) overcome technical limitations in the C++ language.

The first problem is that the reduction algorithm has to find out the most derived common base class of a group of exceptions, and then decide the type for the result-

ing folded exception object. Here again limited reflection support in the C++ language causes problems.

At least a way to find the most derived common base class of two classes is needed. This can then be applied several times to find the most derived common base class for an arbitrary number of classes. In C++, **`dynamic_cast`** provides *run-time* support for asking whether an object belongs to a certain class (either directly or through inheritance). There is no built-in *compile-time* counterpart to **`dynamic_cast`**, but it has been found out that template metaprogramming capabilities of C++ are strong enough to build a compile-time comparison for the inheritance relationship between two types. Such a comparison called `is_base_of` is provided by the Boost Typetraits library [Abrahams *et al.*, 2006] and the C++ TR1 library extension [ISO/IEC JTC1/SC22, 2006, §4.6]. It is also included in the new C++11 standard [ISO/IEC, 2012, §20.6.5]

Unfortunately even `is_base_of` does not solve the folding problem. It allows compile-time checking of inheritance relation between two classes, but those classes have to be known beforehand. The author knows no mechanism in C++ to navigate arbitrary inheritance hierarchies, either during compile-time or run-time. The same limitation caused problems with exception mapping in Section 4.6.3.

The problem is quite similar to the exception hierarchy traversal needed in polymorphic exception mapping in Section 4.6.4. Extra information about inheritance relationships is needed. At least each exception class needs to tell its base classes so that the common most derived base class can be found.

The second problem is caused by the fact that exception *objects* are thrown, not just their types. Even though exception resolution is based purely on the types of exceptions, exception handlers may need the information contained in the exception objects. So it is not enough just to find a common base class for the exception types, there also has to be a way to allow the resulting new exception object to collect necessary data from the individual exception objects.

It is clear that types of individual exception objects cannot be taken into account when their data is combined (exceptions to be folded belong to arbitrary derived exception classes). It is however known that each individual exception object belongs to the chosen common base class. The base class itself knows what kind of common information the exception objects contain on the base class level and can provide a method for folding that information into a single base class exception object.

5.7.2 Providing folding information

Since folding is performed when the resulting base class exception object is created, one option is to do the folding in the constructor of the object. In this thesis, a *folding constructor* of an exception object is a constructor which takes as its parameters a compound exception which contains all the exceptions to be folded.[2] With folding constructors, the exception hierarchy provides enough information to fold an arbitrary number of exceptions in the hierarchy into a single one. Another possibility is to delegate folding to a static member function which returns a pointer to a dynamically created exception object which is the result of folding.

Metaprogramming can be used to let the programmer to choose which one to use. If a static member function `create_folded` exists, it is used to perform folding. Otherwise a folding constructor is used, if present. If neither of these exist, folding to that class is not possible.

Folding inheritance information can be provided in the same manner as mapping inheritance information in Section 4.6.4. In principle the same `MapBases` typedef could be used, but in some exception hierarchies the whole hierarchy should not be used for folding. For example, if the exception hierarchy contains base classes for fatal errors and minor errors, it is probable that fatal errors and minor error should not be folded together to their common base class, losing essential information.

For this reason a separate typedef `FoldBases` is used for inheritance information needed in folding. Its format is similar to `MapBases`. If there is no need to terminate folding to a certain level in the hierarchy, `FoldBases` and `MapBases` can be identical. Otherwise `FoldBases` can omit some base classes if folding to those bases is not wanted. Listing 5.8 on the following page shows an example of an exception class with folding information. In the listing `FoldBases` does not mention the base class `B2L`, so common base class search does not proceed to that base.

5.7.3 Using trait classes for folding information

Folding constructors and embedded `FoldBases` typedefs can be used in user-defined classes, but they are problematic with third party exception hierarchies, where new member functions cannot be added later. An external folding mechanism is needed

[2] Folding constructors resemble the new *initializer-list constructors* in the C++11 standard. [ISO/IEC, 2012, §8.5.4].

```
1  class E1L : public B1L, public B2L { /* ... */ };
2
3  class E1Rpc: public E1L, public RpcException<E1Rpc>
4  {
5  public:
6    // Mapping information
7    typedef E1L MapBases(B1L,B2L)
8    // Folding information (does not fold towards B2L)
9    typedef E1L FoldBases(B1L);
10   // Folding function
11   E1Rpc* create_folded(CompoundException& ce) { /* ... */ }
12 };
```

LISTING 5.8: *Folding information in an RPC exception class*

for those classes. Similarly, information about inheritance should be possible to put outside the class.

Trait classes discussed in Section 4.6.5 can also be used for exception folding. Information and functionality concerning an exception class can be put into its trait class. The default implementation of the trait template can look inside the class so that programmer's own exception classes can use embedded member functions and typedefs.

To create an exception object using folding, its trait class should provide a static member function `create_folded`, which takes as its parameter the compound exception containing exceptions to be folded (all of which are assumed to be derived from the exception class in question). It returns a dynamically created exception object which is the result of folding.

Folding information on inheritance can be provided in the trait class using a `FoldBases` typedef as described earlier. Listing 5.9 on the next page shows an example of a concurrency trait with folding information.

5.7.4 Folding to a single level in inheritance hierarchy

It is fairly straightforward to check whether a set of exceptions belong to a *given* base class, since this check can be done with **`dynamic_cast`**. Even though this simple folding does not navigate the inheritance hierarchy, it is enough for simple cases, where the whole inheritance hierarchy is not of interest. Two reduction class templates `FoldAnySingle` and `FoldAllSingle` are provided for this purpose. The templates take as their parameter the RPC exception class to which folding should be performed, if folding is possible. `FoldAnySingle` reduction folds any exceptions that belong to the

```
1  class E1L : public B1L, public B2L { /* ... */ };
2
3  template<>
4  struct ConcTraits<E1L>
5  {
6   // Mapping information
7   typedef E1L MapBases(B1L,B2L)
8   // Folding information (does not fold towards B2L)
9   typedef E1L FoldBases(B1L);
10  // Folding function
11  ConcExcp<E1L>* create_folded(CompoundException& ce) { /* ... */ }
12 };
13
14 typedef ConcExcp<E1L> E1Rpc;
```

LISTING 5.9: *Folding information in a concurrency trait*

given base class or are derived from it. It succeeds if there is at least one exception of required type. `FoldAllSingle` in turn requires that all exceptions to be reduced are suitable for folding.

The benefit of `FoldAnySingle` and `FoldAllSingle` is that they work for all RPC exception classes as long as the target class provides a folding constructor or a folding function. However, they are not practical for folding complete inheritance hierarchies, since that would need a `FoldAllSingle` for each base class, combined with the `Choose` combinator, for example.

5.7.5 Implementation of full inheritance based folding

When the programmer provides information about the exception inheritance hierarchy (using `FoldBases`), mechanisms for complete folding can be implemented. As explained in Section 4.5, implementing RPC exception hierarchies requires a traditional exception hierarchy and separate RPC classes for each class in the hierarchy. The RPC classes are derived both from their traditional counterpart and template `RpcException` (with the RPC class itself as a template parameter).

Implementation of folding is shown in Listing 5.10 on page 109. The source code is from the KC++ implementation. Folding reduction is implemented by a static member function `fold(CompoundException*)` of base class template `RpcException`. First `RpcException<ERpc>::fold` uses a check similar to `FoldAllSingle<ERpc>` to find out whether all exceptions are derived from `E` (which is found using the return type of the `FoldBases` typedef). If all exceptions of the correct type, an `ERpc` object is cre-

ated using the `create_folded` function or the folding constructor. Otherwise `fold` delegates the task to `BRpc::fold`, which works one step up in the implementation hierarchy (`BRpc` is again found using the `FoldBases` typedef). If all levels fail, 0 is returned. If more than one base class is defined, each base class is tried in the order which they are listed.

The remaining problem is at the beginning of folding reduction, when a correct instance of `fold` must be found to start the folding chain. Since the reduction mechanism has no static information on the types of the exceptions, dynamic typing is needed. For this purpose, `RpcExcpBase` defines a pure virtual function `dynfold(CompoundException*)`. Each `RpcException` instance implements it and the implementation just calls `fold`.

With this virtual function folding reduction can be performed by calling `dynfold` on any of the exceptions to be folded. Then `dynfold` in turn calls `fold` and starts the folding test chain from the level of the exception used. Since folding is searching for a most derived common base class, it cannot be further down in the inheritance hierarchy than any of the exceptions to be folded.

5.8 Implementation issues with multiple exceptions

This section presents some issues in the implementation of concurrent exceptions as well as restrictions and limitations imposed by the standard C++ language. In many cases, these limitations have forced some compromises compared with the ideal way of handling multiple concurrent exceptions.

5.8.1 Keeping track of thrown exceptions

One of the problems is to implement reduction functions and compound exception handling in standard C++ without language changes. When an exception is thrown, C++ code has no control over the program execution until a suitable exception handler is found and its code is executed.

Future groups work by registering themselves with futures. If a future containing an exception is accessed, it first synchronises with the future group, triggering reduction. If the lifetime of a future group ends before any future has triggered synchronisation, the destructor of the group performs synchronisation and if necessary, reduction.

```
1  class RpcExcpBase
2  {
3  public:
4    virtual shared_ptr<RpcExcpBase> dynfold(CompoundException& ce) const = 0;
5  };
6
7  template<typename E, RpcExcpBase::TypeID E_TYPE = NO_E_TYPE>
8  class RpcException : public RpcExcpBase
9  {
10 public:
11   virtual shared_ptr<RpcExcpBase> dynfold(CompoundException& ce) const
12   {
13     return fold(ce);
14   }
15
16   static shared_ptr<RpcExcpBase> fold(CompoundException& ce)
17   { // This implementation is simplified from the real version
18
19     bool folding_possible = true;
20     if (ce.empty()) { folding_possible = false; } // Nothing to fold
21     for (RpcExcpBaseIterator i = ce.begin(); i != ce.end(); ++i)
22     {
23       // FoldBasesAux<E> is a helper metafunction which provides
24       // access to the components of the FoldBases typedef
25       // Orig is the type of the original non-RPC exception class.
26       if (!std::tr1::dynamic_pointer_cast<typename
27                 FoldBasesAux<FoldType>::Orig>(*i))
28       { folding_possible = false; break; }
29     }
30
31     if (folding_possible)
32     { // Folding is possible, create a folded exception
33       shared_ptr<RpcExcpBase> result;
34       if (ce.size() == 1)
35       { // Only one exception, no need for folding
36         result = *ce.begin();
37       }
38       else
39       { // Do folding
40         result = shared_ptr<RpcExcpBase>(E::create_folded(ce));
41       }
42       ce.clear(); // Remove exceptions from reduction set
43       return result;
44     }
45     else
46     { // Folding was not possible on this level, try base class
47       // FoldBasesAux<E>::basefold is a helper (meta)function which
48       // uses FoldBases to call fold on the base exception classes.
49       return FoldBasesAux<E>::basefold(ce);
50     }
51   }
52 };
```

LISTING 5.10: *Implementation of full exception folding*

However, it is possible that the destruction of a future group is triggered by an exception thrown from outside the futures of the group. In this case, the destructor of the future group has no standard way to gain access to the thrown exception object itself. This information is needed by thrown-reduction functions, because they need knowledge about the thrown exception in addition to pending exceptions received through futures. An additional mechanism is needed to provide information about the currently thrown exception.

Concurrent exception handling keeps track of the current exception status by having its own *"throw stack"*. Each execution thread has its own internal stack of currently thrown concurrent exception objects. Exception objects are added to the stack when they are created and their destructors remove the objects when their handling completes and the exception becomes finished.

Exception objects in C++ have no knowledge of when they are actually thrown. Concurrent exception objects make a simplifying assumption that they are thrown immediately after their creation, and the constructor of the exception base class registers every new exception object to the throw stack. Similarly, the destructor of the base class removes an exception object from the throw stack when the exception becomes finished. This way the system can keep track of the most currently thrown concurrent exception object.

When a future group is destroyed (or its synchronisation is explicitly requested), the group first waits for all registered futures to get their values. Then it creates a compound exception object, where all pending exception objects are collected from the futures. If the throw stack is not empty, exception handling is already in progress and thrown-reduction should be performed. The destructor calls the C++ library function `std::uncaught_exception` to find out whether there are any unhandled thrown exceptions. Section 5.8.2 will present restrictions on the result of `std::uncaught_exception`. The future group then takes the topmost thrown concurrent exception object (if any) from the stack. It assumes that this is the latest thrown but not yet handled exception and passes it to the thrown-reduction function. Section 5.8.2 will also discuss situations where this assumption might be incorrect.

5.8.2 Restrictions in the concurrent exception model

Concurrent exception handling in this thesis is implemented using standard C++, without modifying the C++ exception model (or compiler). This means that concur-

rent exceptions have to cope with restrictions caused by the C++ exception model, which has been designed without thinking about issues caused by concurrency.

This section discusses the most important issues and restrictions in the concurrent exception handling model presented in this thesis. It also describes the situations where these restrictions apply and suggests workarounds and ways to avoid the restrictions, if possible.

Changing a thrown exception

The concurrent exception handling model allows programs to handle a situation where several thrown exceptions from concurrent calls end up in a single execution thread as pending exceptions. Reduction functions allow the program to interpret the situation and reduce the number of exceptions. The fact that each concurrent exception object can act as a compound exception means that several exceptions can be grouped together for later analysis and reduction.

One problem with reduction functions and multiple exceptions is that C++ does not allow replacing a thrown exception before it has been caught in a catch clause. If an exception is thrown from outside a future group, the destructor of the group can call a thrown-reduction function, which in turn can analyse and alter the set of exceptions in the group. However, an already thrown exception object cannot be replaced by another, more important exception. This limits the abilities of thrown-reduction functions and restricts their use. A partial solution is to catch the thrown exception and perform necessary reduction in the exception handler, as discussed in Section 5.5.4.

Compound exceptions

If `CompoundException` class is used to represent a situation with multiple exceptions, the program cannot catch exceptions based on their type. A compound exception object is always of type `CompoundException` no matter what exception objects it contains. For this reason `CompoundException` is mainly useful as a temporary container of unreduced exceptions, and it should be replaced by another exception in a reduction function.

Non-RPC exceptions

Exceptions are added to the throw stack in the concurrent exception base class constructor and removed in the destructor. Therefore non-RPC exceptions (i.e. exceptions not derived from the concurrent exception base class) are not added to the throw stack. This means that future groups and reduction functions cannot get access to thrown non-RPC exception objects. Since non-RPC exceptions do not have to have a common base class, referring to those exceptions would be impossible in C++, anyway.

Not having access to these objects mean that information about these exceptions cannot be part of the exception reduction process. Future groups' destructors can detect the presence of a thrown non-RPC exception by using `uncaught_exception` (with certain limitations) and call thrown-reduction instead of normal reduction, but a pointer to the already-thrown exception object cannot be provided to the thrown-reduction function.

In practise, the use of non-RPC exceptions should be minimised in programs using concurrent exception reduction, especially in places where future groups and reduction functions are used. The wrapper template `ConcExcp` presented in Section 4.5 can be used to create concurrent versions of ordinary exceptions. Exception mapping explained in Section 4.6 can be used to wrap and re-throw non-RPC exceptions in concurrent form.

Standard C++ and `std::uncaught_exception`

Some of the mechanisms presented in this chapter use the C++ library function `uncaught_exception`, which was mentioned in Section 2.2.3. The problem is that although `std::uncaught_exception` tells if an exception has been thrown, it does not tell whether the code calling `uncaught_exception` can safely throw another exception. C++ supports multiple simultaneous exceptions as long as they do not end up at the same level during stack unwinding. The return value of `uncaught_exception` just tells whether exception handling on some level is in progress.

Future groups use `std::uncaught_exception` in their destructor to find out if exception handling is currently active, in which case thrown-reduction is called instead of normal reduction. The limitations of `uncaught_exception` mean that if a future group is destroyed in code which is called from the destructor of some class, it is possible that thrown-reduction is called even though normal reduction would

also be possible (the exception thrown by normal reduction could be caught in the destructor of the class).

In practise the only way around these problems is to avoid using future groups in code which is executed as the result of stack unwinding, i.e. destructors or code called from destructors. This limitation should not make programming difficult, since standard C++ practises already warn against destructors performing actions which may fail and throw exceptions.

These problems are the direct result of the way C++ allows stacking of exceptions. There is no static compile-time way of knowing whether it is safe to throw an additional exception during stack unwinding, since there is no static way of knowing whether that additional exception would be caught before it ends up on the same level as the original exception.

5.9 Applicability to other languages

Multiple concurrent exceptions are a problem that remains unsolved in many programming languages in addition to C++ (this is discussed later in Chapter 7). Therefore other programming languages may benefit from multiple exception techniques presented in this chapter.

The multiple exception mechanism is heavily based on futures for asynchronous return value passing. Therefore its applicability to languages without futures is questionable. Fortunately, futures are being added to many modern languages (e.g., C++11 and Java). If futures are provided, the ideas behind compound exceptions, future groups, reduction contexts and reduction functions are not dependent on the programming language.

Synchronising among futures using a future group is easily applicable to other languages. Futures just need to keep track of future groups they belong to and trigger the synchronisation of those groups. Forcing synchronisation before leaving a try block is more challenging. Current mechanism achieves this by having the destructor of the future group perform synchronisation. Since destructors and deterministic destruction is not very common in modern programming languages, alternative mechanisms should be used (ProActive mentioned later in Section 7.3.2, for example).

Exception reduction presented in this chapter should be applicable to other languages as well. It only requires that programmer can register reduction functions to

future groups. This is possible through function pointers, inheritance, lambda functions, etc, thus suitable mechanisms should exist in most modern programming languages. Exception reduction is possible even without futures, if other mechanisms are used to collect together multiple concurrent exceptions before handling them.

The mechanism to combine reduction functions using metaprogramming can be used in other languages depending on the metaprogramming facilities of the language. The mechanism in this chapter relies heavily on compile-time template metaprogramming, but similar results can be achieved through other means, such as run-time reflection, for example. If a language supports functions as first-class citizens, writing reduction combinators should not be difficult.

A special case of exception reduction was presented, where a group of exceptions is folded to a single exception whose type is the most derived common base class of the exceptions to be reduced. This folding reduction can be used in other languages supporting exception hierarchies, if their reflection capabilities are strong enough to traverse and analyse class hierarchies. Since C++ lacks run-time reflection, folding is achieved by providing additional type information through trait classes. In languages with run-time reflection, folding could be achieved more easily by traversing exception hierarchies using reflection. Folding several exceptions to one still needs folding functions which combine information from several exceptions to one. These functions would still have to be provided by the programmer.

In many cases the mechanisms presented in this chapter have been limited by the C++ languages, especially its static compile-time reflection and rigidity of its exception handling. If the mechanisms are applied to other languages, these limitations do not necessarily apply any more. One great limitation in C++ is that it does not allow the program to replace a thrown exception without catching it and throwing another exception. This makes it impossible to perform proper exception reduction in the destructor of a future synchroniser, if an exception has already been thrown. To the knowledge of the author, this limitation exists in most major programming languages, even ones with run-time reflection. For example, Java language provides no better support for accessing or replacing a thrown exception.

5.10 Summary

In this chapter it has been shown how support for multiple exceptions in asynchronous concurrent calls can be added to C++. This capability is achieved with

futures and future sources for asynchronous return value handling, future groups for mutual synchronisation. Reduction contexts, compound exceptions, and reduction reduction functions are used for exception resolution and selection. All this has been achieved using a library-based approach, without changing the C++ syntax or the underlying compiler.

This chapter has also presented a template metaprogramming based approach for combining reduction algorithms from ready-made components. A collection of such components are given, as well as tools for reduction based on exception hierarchies.

The source code for an implementation of the mechanisms presented in this chapter (as well as the source of the entire KC++ library) can be found in `http://www.cs.tut.fi/ohj/kcpp/kcpplib-html/`.

Chapter 6

A case study and evaluation

This chapter contains a case study and evaluation of the exception passing and reduction mechanism described in this thesis. The purpose of the evaluation two-fold: to use mechanisms in a real application to check their feasibility and gain experience on their use, and to provide a platform for performance measurements to verify that the mechanisms do not cause unacceptable overhead.

Before the actual case study, Section 6.1 discusses performance tests used to measure the low-level performance of the RPC exception passing mechanism. Section 6.2 presents a case study using the concurrent exception mechanisms: a concurrent implementation of the Observer pattern. Section 6.3 extends this case study by presenting an application built on the concurrent Observer implementation. The application is a simple concurrent graphical image manipulation program which uses the Observer pattern to notify changes in the image chain. The motivation of this case study was to use concurrent exception handling and exception reduction in a realistic setting to test its expressiveness and find out problems and limitations.

The case study gives an opportunity to measure the performance impact of the mechanisms presented in this thesis. Section 6.4 discusses the performance of the case study. Tests were run on the case study application and these results are analysed and compared with the results given in Section 6.1.

6.1 Performance of RPC exception passing

In C++ throwing an exception is a heavy operation, and many C++ style guides and textbooks emphasise that for this reason exceptions should only by used in really

exceptional situations [Stroustrup, 2000, §14.5][Meyers, 1996, Item 15]. The overhead of exceptions is usually compared with the overhead of a normal function call. In a concurrent environment the situation is different, since RPC calls are much slower than normal function calls, and on the other hand RPC exception passing mechanism imposes additional overhead on top of normal exception handling. For this reason it was regarded necessary to measure the impact of exception handling using the RPC mechanism described in this thesis, and compare it with normal C++ exception handling overhead.

6.1.1 Test setup

Performance of exception passing was tested by repeatedly calling a simple function that returned an integer or threw an exception depending on the test. Measurements were made for three test cases. The first test was performed with no exception handling code and with exception support disabled in the compiler. The second test was performed with necessary exception handling code included but without actually throwing any exceptions. In the third test an exception was thrown from the function each time.

Since exception handling in normal C++ is typically several orders of magnitude slower than normal return, exceptions are often not used in time-critical segments of code. Therefore, it is not reasonable to compare exception handling performance directly to cases where exceptions do not occur. However, measuring the normal return gives an estimate for normal function call overhead.

Each of these three tests was run in three different setups. The first setup used normal C++ function calls and exception handling. The second setup was a modified program that used serialisation for parameter and return value passing, and the mechanism described in this thesis for exception propagation, but within the same address space. The third setup was to use KC++ for a real RPC call between address spaces. KC++ uses active objects executing concurrently in separate address spaces, and futures as an asynchronous communication mechanism. However, only synchronous calls were used in this test to make results easier to compare with each other. The KC++ tests give a realistic estimate for RPC overhead in remote invocation and having to pass serialised data to another address space.

All tests were run under 64-bit OpenSuse 11.0 Linux with kernel 2.6.25.18-0.2 and Intel Core 2 processor running at 2380 MHz. Only one processor core was

enabled in the operating system (to make comparisons fair for both serial and parallel tests). CPU frequency scaling was turned off. GCC 4.3.2 compiler was used with both -O2 and -O3 optimisations.[1] The code calling the test function and the test function itself were placed in separate compilation units in order to make sure that the compiler could not use inlining to optimise the actual function call away. Test results were interpreted using the *K-best* method [Bryant and O'Hallaron, 2003, Ch. 9] — each test program was run N times and its running time was measured. If the relative differences of the K best results were less than ϵ, the best result was accepted as "representative" with little extraneous fluctuation. Parameter values for the K-best criteria were $N = 20$, $K = 3$ and $\epsilon = 0.01$ (these values are from [Bryant and O'Hallaron, 2003]).

The results are shown in Table 6.1 on the next page. The first column in the table tells how exceptions were handled in the test. Since returning from a function and throwing an exception is a very fast operation, each test program contained a loop which performed the operation repeatedly. The second column shows how many times the test loop was run, and then third column shows the total time, both for -O2 and -O3 optimisations. The number of tests for each run was chosen so that each individual test run lasted approximately 10-60 seconds. This was necessary to keep the total time of the whole test run within reasonable limits.

The last column is the most important and shows the time consumption of one individual call, again for both optimisation flags. Results in the last column were rounded to two significant digits, because K-best criteria for accepting the results was $\epsilon = 0.01$ (i.e., 1 % of fluctuation was allowed). These results are presented as a logarithmic graph in Figure 6.1 on the following page.

6.1.2 Analysis of test results

Tests 1, 4, and 7 were performed with exception support turned off in the compiler. They are meant to represent the base line on top of which exception handling overhead is added. Tests 2, 5, and 8 were performed with exception support on, but without throwing any exceptions. Finally tests 3, 6, and 9 show the case where the function returned by throwing an exception instead of returning a value. Test 6b is otherwise the same as test 6a, but the client first fetches the exception and then throws it separately, instead of asking the library to throw the exception right after

[1] Both optimisations are included since the tests indicate that -O2 can be more efficient than -O3 in certain situations.

	Test	# of calls	Time -O2/-O3 (s)	Time/call -O2/-O3 (µs)
1.	No exception handling	10^{10}	25.24/29.45	*0.0025* / *0.0029*
2.	Excp handling, no throw	10^{10}	29.45/29.45	*0.0029* / *0.0029*
3.	Excp handling and throw	10^{7}	47.53/47.67	*4.8* / *4.8*
4.	Serialisation, no excp handling	10^{8}	31.16/34.21	*0.31* / *0.34*
5.	Serialisation, no throw	10^{8}	31.79/34.49	*0.32* / *0.34*
6a.	Serialisation and throw	10^{6}	16.77/16.67	*17* / *17*
6b.	Serialisation, separate throw	10^{6}	19.70/19.73	*20* / *20*
7.	KC++, no exception handling	10^{6}	9.15/9.22	*9.2* / *9.2*
8.	KC++, excp handling, no throw	10^{6}	9.39/9.55	*9.4* / *9.6*
9.	KC++, excp handling and throw	10^{6}	32.77/32.79	*33* / *33*

TABLE 6.1: *Results of performance tests*

unmarshalling. This is closer to the normal KC++ behaviour where an exception is first unmarshalled to a future and then thrown.

Results 1–2 and 4–5 show that as many compiler writers claim, exception handling (try-catch blocks with exceptions enabled in the compiler) does not affect performance as long as exceptions are not actually thrown. For some reason the GCC compiler produced a faster program with -O2 optimisation and exception handling turned off, but with -O3 optimisation both versions performed similarly. The reason for this compiler optimisation peculiarity was not investigated further.

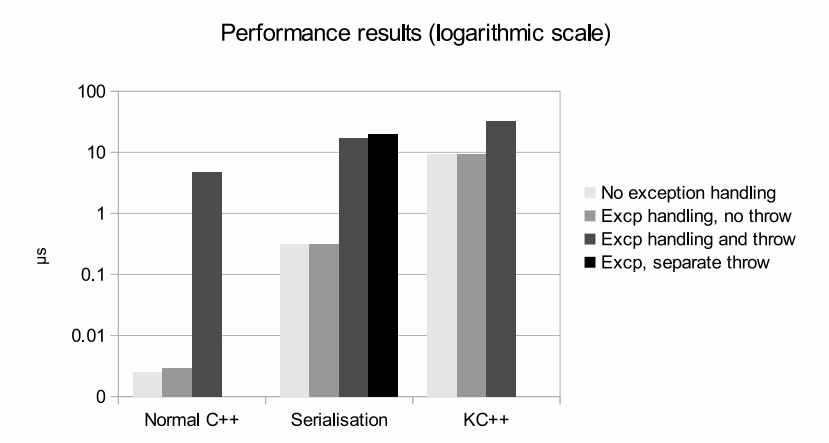

FIGURE 6.1: *Test results in graphical form*

Results 1–3 show that throwing and catching an exception in this test is three orders of magnitude slower than passing a normal return value. This result is similar to those mentioned elsewhere: “Throwing an exception may be as much as three orders of magnitude slower” [Meyers, 1998, Item 15]. Exception handling in modern C++ compilers does not cause performance penalties when exceptions are not thrown, but this results in a heavy overhead when an exception is thrown: “All associated run-time costs occur only when an exception is thrown. However, because of the need to examine potentially large and/or complex state tables, the time it takes to respond to an exception may be large, variable, and dependent on program size and complexity.” [ISO/IEC WG21, 2006].

Result 6a shows that using the exception serialisation mechanism described in this thesis causes exception handling to perform over 3 times slower than with normal exception handling. This increase is as expected, since serialisation inevitably means that the exception must be thrown twice, once in the test function and another time in the caller after unmarshalling. The remaining overhead is explained by serialisation.

Result 6b shows that when the exception object in the caller is first created dynamically and thrown later (which is what happens with futures), exception handling is 4 times slower than in normal C++. This is not surprising since dynamic memory allocation and shared pointers cause additional overhead. In addition to this, in test 6a creating and throwing the exception in the same function allows more optimisation possibilities for the compiler.

Comparing result 6b with results 4–5, which show the overhead of serialisation, the performance cost of an exception is quite acceptable, since it is only 60 times slower than normal serialised return, compared with the 1600 times slower with no serialisation (result 3 compared with 1–2). This can also be seen from the logarithmic graph in Figure 6.1, which shows that relative cost of exceptions decreases when the cost of normal return increases.

Results 7–9 show performance tests implemented using the exception mechanism in KC++. The test program consisted of two (active) objects, with one object synchronously calling a method of the other object. The method either returned a normal return value (7–8) or threw an exception (9).

KC++ uses POSIX message queues for communication between processes, including method calls, their parameters, return values, and exceptions (the latter two through futures). Since the caller and callee execute in separate processes, even

synchronous calls include context switching in the operating system. This makes the measurements fluctuate much more than in the single-threaded experiments. Therefore the results 7–9 should not be interpreted as precise timing values. However, they show that adding exception support to the system causes no substantial performance penalty when exceptions are not thrown (tests 7–8).

Propagating an exception in KC++ takes 3.5 times longer than returning a value. Compared with the earlier 60 times difference this may seem to be low, but it is explained by the much greater overhead caused by message queues and process dispatch. The additional overhead of exception handling is 24 μs for KC++ and 20 μs for the simple serialisation test, so they are close to each other. This means that the mechanism described in this thesis adds little overhead to exception handling in an RPC system, relative to the cost of communication.

6.2 Case study: concurrent observer

In order to evaluate exception handling mechanisms in this thesis, a realistic case study is needed. The case study presented here demonstrates how the mechanisms described in this thesis can be used and tests their applicability in the real world.

The aim was to get experience on the usability of the concurrent exception mechanisms and to test their expressive power and find out their limitations. One goal was also to build a case study which could be run both concurrently with exception reduction, as well as sequentially using traditional exceptions. This allows a reasonable performance comparison between the two approaches, which is presented later in Section 6.4.

6.2.1 Structure of the case study

For the case study, a known design pattern was implemented and then an application was written using the pattern framework. Implementation of a design pattern provides a generic case study, and it also tests that the approach is able to handle a program which consists of several encapsulated modules (i.e., the generic design pattern code and the actual application code).

The selected design pattern was Observer [Gamma *et al.*, 1996, Ch. 5]. The reasons for selecting this design pattern were the following:

- It is widely used (for example, to loose the coupling between the model and the view in the Model-View-Controller (MVC) model [Reenskaug, 1979], which in turn is used in graphical user interfaces, and became well-known by the Smalltalk user interface framework [Krasner and Pope, 1988].
- There are obvious benefits from adding concurrency to Observer. In the Observer pattern observers are notified when a subject changes its state, and these notifications can be concurrent since they do not depend on each other.
- The pattern itself contains many possible sources of exceptions, and handling of these exceptions is non-trivial and has several alternatives.

First the Observer pattern is discussed in terms of concurrency and exception handling. Then the pattern is implemented as a small framework of base classes from which applications derive concrete classes using the Observer pattern. After that found concurrency and exception handling issues are discussed and analysed.

6.2.2 The Observer pattern

The Observer pattern provides the roles of *"subject"* and *"observer"*. Observers can register themselves to one or more subjects. When the state of the subject changes (an *update*), the subject *notifies* all its registered observers. The notified observers can then ask the subject about its modified state and update their own state accordingly.

Structure of the Observer pattern is shown in Figure 6.2 on the next page. The base class Subject contains all necessary operations of subjects (registration and notification). Real subjects are derived from the base class and they contain the actual state and methods needed for changing and inspecting the state. Similarly, the base class Observer contains the notification interface through which subjects notify their observers. Real observer classes are derived from the base class and they implement the notification method.

6.2.3 Concurrency in the Observer pattern

The Observer pattern described in [Gamma *et al.*, 1996] does not discuss concurrency in relation to the Observer pattern. Since the pattern is quite simple, the only obvious place where concurrency has some benefit is the notification. Concurrency

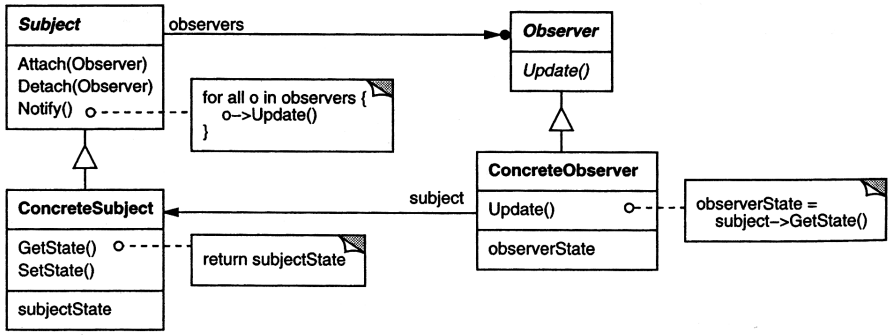

FIGURE 6.2: *Structure of the Observer design pattern (from [Gamma* et al.*, 1996])*

also presents some new problems and challenges to the implementation of the Observer pattern. Edward Lee presents Observer as an example of thread concurrency problems in [Lee, 2006].

Lee points out that when `Notify()` is called on a Subject, in a concurrent environment it can send notifications to Observers asynchronously. In addition to this, the `GetState()` and `SetState()` methods in ConcurrentSubject can also be asynchronous, as well as `Attach()` and `Detach()`. If the used concurrency model allows several execution threads in one object, attaching and detaching observers while notifications are being invoked can lead to problems. For this reason mutual exclusion needs special attention. It is important that `Attach()`, `Detach()`, and `Notify()` in a Subject cannot disturb each other. Similarly having several `SetState()` methods running concurrently is potentially dangerous.

Another possible source of interference is calling `SetState()` while notifications are in progress, causing a state change in the middle of notification. This may result in some observers querying a different subject state than others. This issue arises also in non-concurrent observers and is further discussed in the next section.

6.2.4 Sources of exceptions in the Observer pattern

The *Design Patterns* book discusses several possible error situations in the Observer pattern. Further exceptional situations are discussed in [Szyperski *et al.*, 2002] and [Gruntz, 2002], for example. This section briefly discusses possible sources of exceptions in Observer.

Dangling references to deleted subjects (and observers). Since part of the motivation behind the Observer pattern is to separate Subjects and Observers, the lifetime of these objects is not necessarily mutually dependent. This situation is already mentioned in [Gamma *et al.*, 1996]. Since observers and subjects may be destroyed independently, references from observers to subjects and vice versa must be able to handle a situation where the other party has already been destroyed. A somewhat similar (but much simpler) error situation is a case where a null pointer is registered as an observer.

Adding or removing an observer during notification. Adding or removing observers while a notification is in progress has to be taken into account whether or not the Observer implementation is concurrent. Updating data structures containing observer references can be fatal if iterations over the same data structure are in progress at the same time. Even if data structure integrity is taken care of, the Observer implementation must decide whether observers added in the middle of notification receive the ongoing notification or not. These problems, together with thread safety issues, are discussed in [Goetz, 2005].

Re-entrant state changes. When a notification occurs, it may be important that each observer observes the same subject state. If a notified observer changes the state of the subject during notification, this property is violated. Even more importantly this means that another notification should be started while the old notification has not yet been completed. This problem of re-entrant state changes is discussed in [Szyperski *et al.*, 2002] and [Gruntz, 2002]. In a concurrent environment this problem is emphasised, since changing the state of a subject may be attempted by any execution thread, not just the notified observer. Normal mutual exclusion mechanisms can easily make sure that two concurrent subject changes are not possible, but if all observers must observe the same subject state, new subject state changes cannot be allowed before possible notifications have been completed.

Cancelling a notification. In a sequential Observer implementation, the notification process proceeds from observer to observer until it has been completed or until an exception occurs in an observer. Only then the thread of execution returns to the subject. If the subject and observer execute concurrently, the subject may want to cancel a notification which has been started but whose observers have not yet

completed their notification procedures. One reason for cancelling a notification is a new state change which makes the notification redundant.

6.2.5 Concurrent Observer using KC++

This section describes the implementation of concurrent Observer pattern using KC++ and the concurrent exception handling techniques described in this thesis. This Observer implementation is then used to implement an interactive image processing program. These implementations are discussed and measured as a case study of the exception handling mechanisms.

The source code of the Observer implementation (as well as the source for the application discussed later in this chapter) can be found in
`http://www.cs.tut.fi/ohj/kcpp/observer-html/`.

Structure of the solution

The KC++ concurrency model is based on active objects with one thread of control. Methods are always run to completion unless voluntary yielding is used. These properties affect how a concurrent Observer can and should be implemented. During notification, calls to `update()` methods in observers can be done asynchronously, resulting in concurrent execution of updates. If exceptions are thrown from `update()`, these exceptions must be analysed and reduced after all updates have completed. The result of the reduction then determines the result of the notification.

Figure 6.3 on the facing page shows the basic structure of the implementation. It closely follows the original GoF structure (Figure 6.2), but has additional methods to handle re-entrant state changes, exceptions and cancellation of operations. Most methods with no return value now return `Future<void>` to enable returning exceptions from asynchronous method calls.

New observers can be added to a subject using `add_observer()` and removed with `remove_observer()`. When the state of a subject changes, the subject calls `notify()`. This method calls asynchronously `update()` on each registered observer and stores the resulting futures. Finally `notify()` schedules `notify_ready()` to be executed when all notifications have completed. This method marks that the notification is complete and performs exception reduction on the `update()` results. Finally `notify()` returns the eventual return value of `notify_ready()` (using a future).

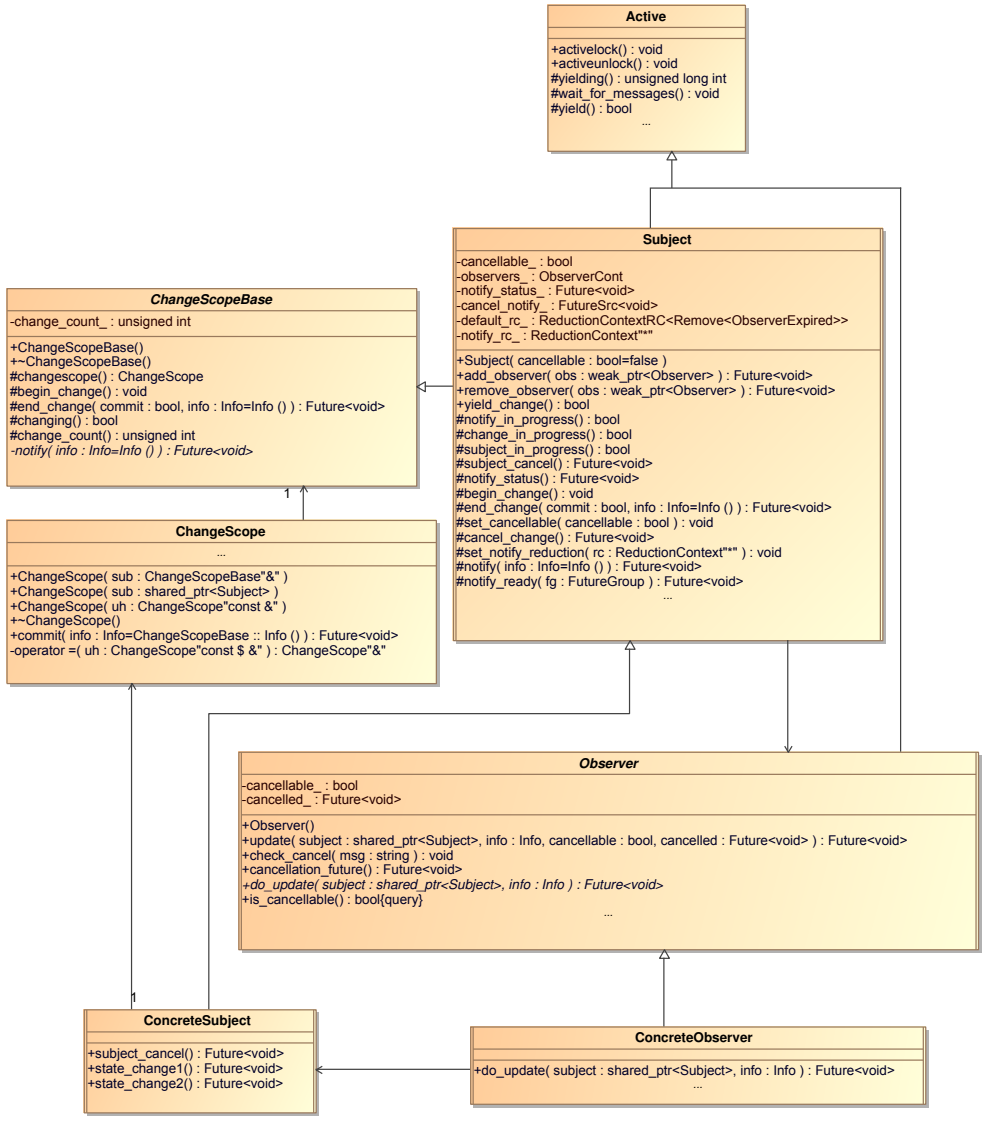

FIGURE 6.3: *Structure of the concurrent Observer implementation*

Handling observers in subjects

Earlier sections mention two possible problems related to observer handling: dangling observers or subjects and addition or removal of observers during notification. Both of these problems are very easy to solve using the KC++ active object model.

The dangling observer problem is solved by storing only *weak pointers* (see Section 3.2.4) to observers in subjects. This way observers can be destroyed regardless of whether they are registered to subjects (weak pointers are not enough to keep an object alive). In `notify()` these weak pointers are converted to strong pointers in order to call `update()`. If an object has expired, this pointer conversion throws an exception, which is caught and stored as the "result" of that particular notification.

A dangling subject pointer problem arises if observers store pointers to subjects and subjects expire before observers. This problem does not occur in the implementation presented here, since observers receive the subject pointer as a parameter to `update()`, so they do not have to store this pointer. The pointer passed to `update()` is a strong pointer, so it keeps the subject alive for the duration of notification. Passing the subject pointer as a parameter also makes it possible to register the same observer to several subjects, and still be able to distinguish subjects in `update()`.

Stacking state changes

The *Design Patterns* book [Gamma *et al.*, 1996] mentions that there are two options for triggering a notification. Either each state change operation automatically triggers a notification, or the performer of the state changes is required to explicitly call `notify()` after all state changes have been performed. The first option is more automatic and secure, the second avoids intermediate notifications between state changes.

Quite many of the potential problems with the Observer pattern concern the handling of state changes and the resulting notification. Notification should only be performed when the state change is complete, even if the state change consists of several steps (smaller state changes). This problem is already mentioned in *Design Patterns*. As a solution to the problem, [Szallies, 1997] introduces *state change scopes*. The idea is to add two additional functions to the the subject interface to mark the beginning and end of a state change. In this implementation, these methods are called `begin_change()` and `end_change()`.

These methods keep a count of started state changes. When `end_change()` is called, notification is started only if there are no further state changes still in effect. This keeps the amount of notifications at minimum, since only one notification is initiated even if a state change is internally implemented by calling several component state changes.

It is also mentioned in [Szallies, 1997] that C++ allows partial automation of marking the beginning and end of state changes. Instead of calling `begin_change()` and `end_change()` manually, this responsibility is delegated to a class called `ChangeScope`. Its constructor calls `begin_change()` and the destructor calls `end_change()`. State changes can be written by putting the state changing code into its own C++ scope and creating a local `ChangeScope` variable in the beginning of that scope. The construction and destruction of the variable then automatically signals the subject about the beginning and end of state change.

This idiom follows the common C++ RAII (Resource Acquisition Is Initialisation) principle. Figure 6.4 on the next page shows an example sequence diagram of a state change consisting of two individual state changes.

State change scopes should be used in each state changing method in concrete subjects. A method starts by creating a `ChangeScope` variable and then proceeds with the actual state change. Even if the state change is implemented by calling other state changing methods, counting in `begin_change()` and `end_change()` triggers the notification only at the end of the outermost state change method.

State change scopes can also be used in the same manner in the users of concrete subjects. If a user wants to make a state change consisting of several calls to a concrete subject, a program scope with a local `ChangeScope` variable is written, and calls are made inside that scope. Notification is started only after program execution leaves the scope.

`ChangeScope` can also be used to make exception handling easier. For this, method `commit()` is added to `ChangeScope`. This method is called at the end of a successful state change, and it in turn calls `end_change` with a parameter representing success. If a state change fails with an exception, `commit()` is not called, and the `ChangeScope` object is destroyed. In this case its destructor calls `end_change()` with a parameter representing failure. This way `ChangeScope` can be used to notify subject about unsuccessful state changes.

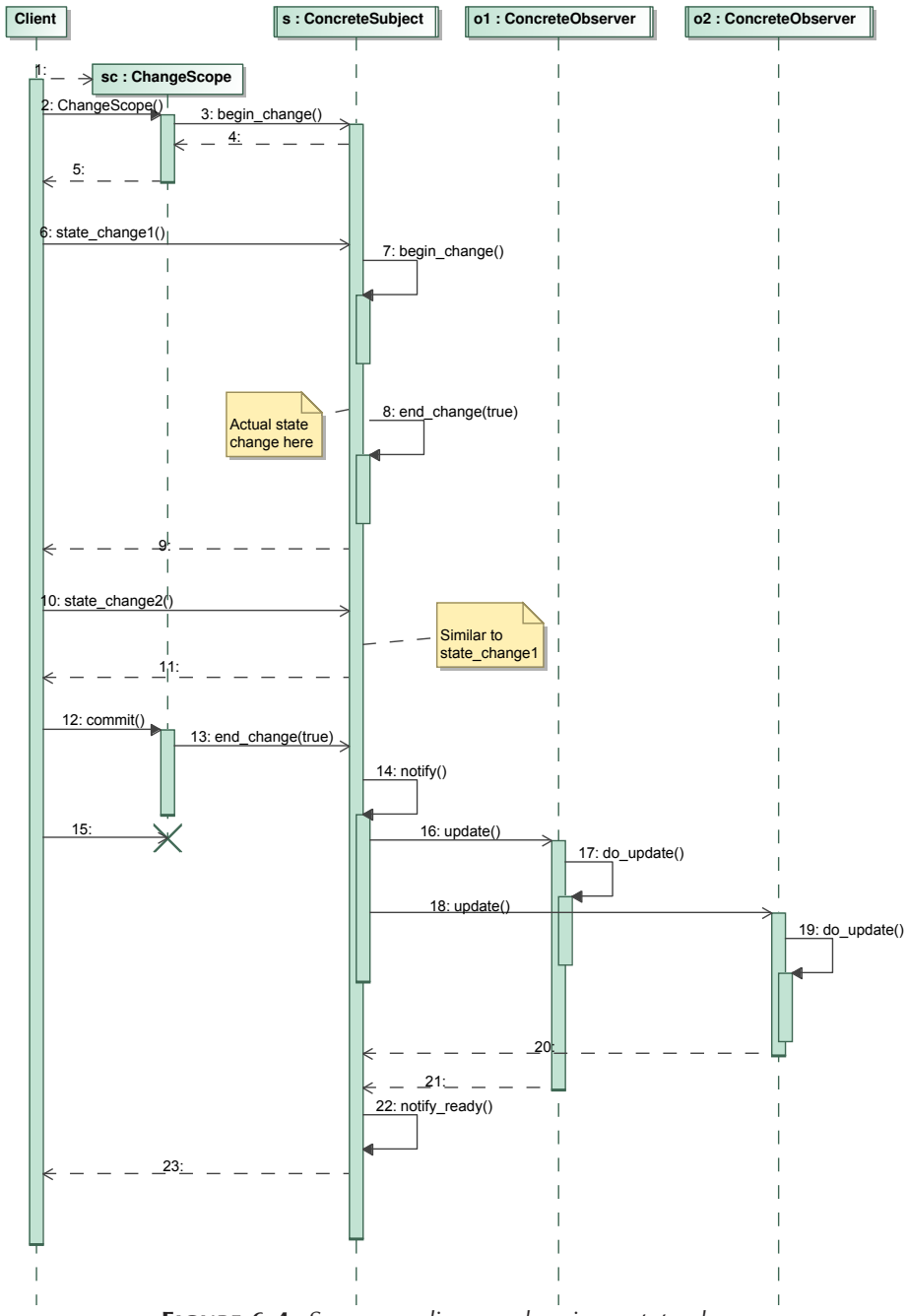

FIGURE 6.4: *Sequence diagram showing a state change*

6.2.6 Concurrency issues in the KC++ implementation

This section discusses concurrency issues in the Observer pattern and how they were solved in this case study.

Mutual exclusion of state changes

In the Observer pattern subjects and observers go through distinct states. Subjects are either "idle", in the middle of a state change, or notifying their observers. Similarly observers are either "idle" or updating their state. A new state change in the subject can only be initiated when the subject is in its idle state.

If the implementation of Observer is sequential, the structure of the pattern defines how these states mix. Observers are in the updating state only when their subject is in its notifying state. Because the program only contains one thread of execution, overlapping state changes are possible only if observers try to initiate a state change from their update methods. Otherwise program execution proceeds through the state changing code to the notification code, performs updates in observers, and then returns the subject to its idle state.

If Observer contains concurrency, several independent execution threads may try to initiate state changes at the same time. In addition to this, if observers perform their updates asynchronously, the execution of notification code in the subject is no longer tied to observer updates. These facts must be taken into account so that the expected functionality of the Observer pattern is not broken.

Only allowing one state change to be active at one time is a fairly normal mutual exclusion requirement, so it is not a problem specifically with the Observer pattern. How this mutual exclusion is achieved depends on the way concurrency is controlled. In this case study, the KC++ active object model only allows one thread of execution in each active object.

If a subject state change is performed by calling a single method of the subject, mutual exclusion is never a problem in KC++ since other methods are not executed while method execution is in progress. However, it is possible that a state change consists of several method calls, using `ChangeScope` or `begin_change` and `end_change` to mark the beginning and end of state change. This would allow other users of the subject to start their state changes in the middle of an ongoing state change.

To prevent this, `begin_change` *locks* the active object to execute only methods from the caller of `begin_change`. This ensures that during the state change method

calls from other sources are not executed. After the state change is complete, the outermost call to `end_change` unlocks the subject again. This makes sure that calls from other clients are not executed before the state change is complete.

An example of a state change sequence

Listing 6.1 shows an example of a state change consisting of several calls to a subject. First a `ChangeScope` object is created. Its constructor calls `begin_change()`, which in turn checks that the subject is idle (and throws an exception if that is not the case) and locks the subject for this caller.

A future group is then created for exception reduction. After this individual state changes are performed and their return statuses are added to the future group. At the end of the scope, the future group synchronises with all the futures. If exceptions have occurred, the future group performs exception reduction and throws the resulting exception. In this case program execution leaves the whole function and the `ChangeScope` object is destroyed. Its destructor calls `end_change` and signals that the state change was not completed. Notifications are not sent.

If the state change calls did not cause exceptions, the future group synchronises with the futures and is destroyed. Then `commit()` of the `ChangeScope` object is called. This calls `end_change` and signals a successful state change. Notification is started and its eventual result is returned in a future. The code in Listing 6.1 waits for this result and the completion of notification. If the notification resulted in an exception, it is thrown on line 10.

```
1  void multiple_change(shared_ptr<MySubject> s)
2  {
3     ChangeScope cs(s);
4     {
5        FutureGroup fg; // Possibly with a suitable reduction context
6        fg.add(s->change1("a"));
7        fg.add(s->change2("b"));
            ⋮
8        // At the end of scope, futures are synchronised, exception may result
9     }
10    cs.commit().wait(); // Commit to change, start notification
11    // At the end of scope, change is terminated
12    // If commit() was not called (exception), notification is not started
13 }
```

LISTING 6.1: *Using `ChangeScope` and a future group for multiple state changes*

This example shows how ChangeScope objects and future groups can be used to perform a state change consisting of several calls to a subject. If exceptions occur during the state changes, the state change is abandoned without notification. The subject is locked for this caller for the duration of the state change, and the end result of notification is propagated from the observers (through exception reduction) to the code which called the state change methods.

The example uses a future group and asynchronous calls to perform all individual state changes before checking the results in the destructor of the future group. Since KC++ only has one execution thread in the subject, all the individual state changes are performed sequentially despite asynchronous calls. An alternative to this is shown in Listing 6.2.

In this version, the code waits for the completion of each state change before continuing, resulting in synchronous operation. If a state change throws an exception, that exception immediately causes termination of the whole state change and the rest of the state change methods are not called. This is different from the first code example where all state changes were always attempted and if exceptions occurred, exception reduction determined the nature of the exception thrown. The Observer pattern implementation presented in this thesis allows both of these approaches, and the programmer may choose an appropriate one for each situation.

Yielding during state changes

Sometimes a single state change operation may require a long time to complete (if the state change requires a lengthy computation). In this case, it is reasonable for

```
1 void multiple_change2(shared_ptr<MySubject> s)
2 {
3   ChangeScope cs(s);
4   s->change1("a").wait();
5   s->change2("b").wait();
        ⋮
6   cs.commit().wait(); // Commit to change, start notification
7   // At the end of scope, change is terminated
8   // If commit() was not called (exception), notification is not started
9 }
```

LISTING 6.2: *Using ChangeScope for multiple state changes without a future group*

the state changing method to voluntarily use KC++ yielding mechanism. This allows other clients of the active object to access the object.

In this case study, voluntary yielding can be done by calling `yield_change()` in the subject. This method first checks that a state change is in progress, unlocks the object and then calls the KC++ function `yield()` to execute possibly pending method calls. After yielding `yield_change()` re-locks the active object.

Voluntary yielding complicates the situation because it explicitly allows other clients of the active object to access the object in the middle of a state change. As always, it is the responsibility of the yielder to make sure that the subject is in a consistent state before yielding. In addition to this, initiating new state changes cannot be allowed while yielding. If a state change is attempted during yielding, `begin_change()` notices that an earlier update has been yielded and throws an exception.

When observers have been notified but have not yet completed their updates, the situation is similar to yielding. Observers should not be allowed to trigger additional state changes in their `update()` methods, because that would cause either additional notifications while old notifications have not completed, or at least observers whose `update()` has not yet been called would see a different subject state than earlier observers.

This problem is solved like the yielding problem. When notification is started, the subject marks its state as "notifying". This state is changed to "idle" only after all notifications have completed. If a state change is attempted during notification, `begin_change()` notices that a notification is in progress and throws an exception.

Cancelling an ongoing notification

Calls to observer `update()`s during subject notification are asynchronous. However, the subject cannot control how long the updates take. As described earlier, the subject cannot start a new state change while notification is in progress, since all the observers must observe the same subject state that caused the notification. For these reasons it would be useful if subjects could cancel an ongoing notification, freeing the subject for new state changes.

KC++ has currently no mechanism for explicitly cancelling the execution of a method of an active object. In fact, forcefully terminating a method execution from outside is problematic, since the active object may be in the middle of a state change

when the method is terminated. Different possibilities for cancellation and their pros and cons are discussed in [Sutter, 2008].

A co-operative solution using futures was chosen for the Observer implementation. When a subject calls observer `update()`, it passes an unbound *cancellation future* as a parameter. This future is created from a future source inside the subject.

The cancellation future acts as a flag for requesting cancellation. If an observer update is a lengthy process, the update code in the concrete observer should check the status of the cancellation future periodically by calling the method `check_cancel` implemented in `Observer`. If cancellation has been requested, this method throws an `UpdateCancelled` exception. This is similar to "Ask Politely" policy in [Sutter, 2008].

The `Subject` class provides a method `subject_cancel()` which binds the cancellation future source, signalling all updating observers. The `subject_cancel()` method returns a future which becomes ready when the notification has finished. This way callers of `subject_cancel()` know when the cancellation (or the normal termination of notification) has happened.

If cancellation is never necessary for some concrete subjects, the constructor of the `Subject` class has a boolean parameter for switching the mechanism off.

6.2.7 Exceptions in the concurrent Observer implementation

Section 6.2.4 presented sources of exceptions in the Observer pattern. This section discusses those exceptions in relation to the case study Observer implementation. It also presents some new exception sources which were noticed during the implementation of the case study.

There are two kinds of exceptions that an implementation of the Observer pattern must take into account: exceptions originating from the pattern implementation itself (i.e., errors and problems in using the pattern), and exceptions coming from outside the pattern implementation, especially from concrete subject and concrete observer classes.

Exceptions of the former kind are easier since they are known to the Observer implementation. The latter kind exceptions, however, can be of any type and be the result of a situation unknown to the Observer. Having to handle unknown exceptions is problematic especially for exception reduction.

Exceptions thrown by the Observer implementation itself

Exceptional situations inside the Observer implementation can be divided into two categories. Some are errors caused by precondition violations when using the implementation (i.e., bugs), while some are problems which can also happen when the implementation is used as specified. Errors of the first kind should not happen, so a reaction to them could be either to throw an exception or to terminate the program using an assertion or some other mechanism [Sutter and Alexandrescu, 2005]. Situations of the second kind can happen in a correctly functioning program, so exceptions are clearly a correct response to them.

Since exception handling is an important part of this case study, it was decided that precondition violation errors also throw exceptions. This way the number of different exceptions in the case study is increased.

The Observer implementation used in this thesis reacts to the following precondition violations by throwing an exception. This list omits some trivial cases like null pointer checks.

- **Observer registration errors.** An observer can only be registered to a subject once, and it is impossible to unregister an observer which has not been registered. In both situations an exception of type `ObserverRegistrationError` is thrown. Since notifications start all updates atomically, addition or removal of observers during notification is not an error.

- **Cancellation errors.** If a subject was created with no notification cancellation capabilities, cancellation is not possible. If this is attempted, an exception of type `CannotCancel` is thrown.

- **Yielding without a state change.** Method `yield_change()` can only be called when a state change is in progress. If the method is called when this is not the case, an exception of type `NotChanging` is thrown.

The following sources of exceptions are not precondition violations, but exceptional situations that can occur during normal program execution.

- **Expired observers.** Registered observers are stored in a subject using weak pointers. This allows observers to be destroyed when they are still registered to a subject. If an expired observer pointer is found during notification, the subject adds an exception of type `ObserverExpired` to the set of exceptions

resulting from notification, just as if `ObserverExpired` had been thrown from an observer update.

- **Cancellation of updates.** If a notification was cancelled and the update of a concrete observer called `check_cancel()`, a `NotifyCancelled` exception is thrown. (The observer may allow this exception to propagate from the update method to the subject.)

- **State change in progress.** New state changes cannot be allowed while notification is in progress or if an ongoing state change has been temporarily yielded. If `begin_change()` is called in such a situation, an exception of type `NotifyInProgress` or `ChangeInProgress` is thrown.

- **State change cancellation.** If a lengthy state change operation yields allowing other methods to be executed, it would be useful to ask for the cancellation of the ongoing state change. The `Subject` class in this implementation has a method `cancel_change()` for this purpose. By default this method does nothing, but concrete subjects may implement it appropriately to cancel an ongoing state change.

Handling and reducing exceptions from outside the Observer pattern

Concrete observer updates are the only place in the Observer pattern where exceptions from outside the pattern code affect the behaviour of the pattern. Since the update methods can throw arbitrary application dependent exceptions, Observer pattern code cannot know in advance what exceptions to expect. On the other hand, if observer updates throw exceptions, these exceptions should be forwarded to the concrete subject which triggered the notification.

When notification is performed in the subject, observer updates are called asynchronously, possibly resulting in multiple exceptions. Since these exceptions originate from the concrete observer, notification code in the pattern implementation cannot contain proper reduction functions for these exceptions. One option would be to embed all exceptions in a compound exception and throw that out of the notification code. This would force the code responsible for the subject state change to catch the compound exception and do necessary exception reduction. Using compound exceptions would hide the types of the exceptions.

For these reasons this case study implements the exception reduction already in the notification code. Concrete subjects can use method `set_notify_reduction` to register their own reduction context object, which is then used to reduce exceptions raised during notification. This mechanism allows the subject notification code to reduce exceptions, but the concrete subject still controls how the reduction is performed. If the concrete subject wants a compound exception containing the exceptions, it can register an empty reduction context with no reduction functions.

By default the subject uses a reduction context which removes all exceptions caused by expired observers but embeds others in a compound exception. This default strategy was more or less arbitrary and was chosen purely as an example. In reality exception reduction depends entirely on the set of possible exceptions and their relationships, as well as the relationship between concrete subjects and concrete observers. Therefore no single default reduction is enough for all cases.

Reduction contexts provided by concrete subjects solve the problem of *how* reduction is performed. A remaining problem is *when* and *where* to perform reduction. The most logical place for reduction would be at the end of the notify method after the results of observer updates have been received. In KC++, an active object can only be executing one method at a time. In this case it would mean that subject is locked for the duration of notification, which would mean that it could not react to queries from observers. The subject must be able to execute methods between calling observer updates and doing reduction based on the results.

The result of notification depends on the results of reduction, which leaves two options. The subject can continuously yield in the middle of notification, allowing other methods to be executed until all observers have completed their updates. New state changes cannot be initiated while the notification is in progress, so there is no risk of "stacked" notifications. Another possibility is to perform reduction in another method and schedule this method to be executed when results from all observer updates are available. The KC++ method scheduling mechanism returns a future containing the eventual return value of the method, so this future can be returned as the return value of notification.

The latter option was chosen in the case study implementation. After calling the updates of all observers, the notification method stores the resulting futures in a future group and then schedules the method `notify_ready()` to be executed when all futures in the future group are ready. This method performs exception reduction and binds the notification future to the resulting exception, if any.

State changes and exceptions

Exceptions can occur in the middle of state changes. The `ChangeScope` idiom makes sure that `end_change()` is called even if a state change is interrupted by an exception. This kind of finalisation after an exception is common to most RAII based mechanisms.

A more interesting question is what the subject base class should do if a state change is interrupted by an exception. Clearly, an exception means that the state change did not succeed as planned. If the concrete subject provides commit-or-rollback semantics, the state is not changed if an exception occurs. Even if the subject cannot revert to its original state, a notification should not be sent automatically if state change ended in an exception. This gives the concrete subject a chance to handle the exception and possibly set the subject to a valid state, after which notification can be sent.

The problem can be solved by giving `end_change()` a boolean parameter *commit* which tells whether the state change was successful. The outermost `end_change()` only starts notification if the value of commit is true. Now state changes can cancel unsuccessful operations without starting notification. This makes state changes somewhat similar to *transactions,* except that roll-back to the original state is not automatic when a state change fails.

The value of the commit parameter is only important for the outermost state change and is ignored for the others. This was considered appropriate for the following two reasons: First, if an inner state change fails and causes the whole state change operation to fail, the outermost state change must be aware of this and also signal failure. Second, it is possible that an upper level code reacts to the failure and manages to resolve the problem, in which case the whole state change still succeeds and notifications should be sent normally.

Observer updates and exceptions

If exceptions occur during state changes, it is quite clear that those exceptions should be thrown to the client which started the state change. On the other hand, if exceptions are thrown from observers' update methods during notification, it is not self-evident where they should be handled [Ploski and Hasselbring, 2005].

One goal of the Observer pattern is to separate observers from subjects. In this respect observers are independent objects who only want to be informed when the

state of the subject changes. If observing that state change causes an exception in an observer, should that exception be thrown to the subject? If yes, it means that the subject must be able to cope with exceptions resulting from observing the subject state. The subject can also forward exceptions to the client who performed the state change, but this means that clients in turn must be able to cope with all observers' exceptions.

The implementation in this case study uses a straightforward approach and propagates all exceptions from observer updates back to the subject which invoked those updates. This approach was chosen for two reasons. First, it provides a good place for exception reduction, since a change in a single subject may invoke several observer updates asynchronously. Second, it still leaves the door open for more complicated approaches. For example, if concrete observers contain their own method for handling exceptions from updates, the update method of the concrete observer can have a try-catch block which catches exceptions and delegates them to the appropriate method.

In this Observer implementation observers' `update()` returns a void-future which contains the possible exception from `update()`. When the subject has received results from all updates (using a future group), it waits until all the futures in the future group are ready (i.e., observers have finished their updates). If exceptions have occurred, the subject reduces the exceptions and throws the result back to the object which initiated the state change.

6.2.8 Discussion on concurrent exception handling

This case study confirms and reveals several things about concurrent exception handling. First, it shows that concurrent exception handling mechanisms presented in this thesis can be used in real code to handle exceptions arising from real world scenarios.

The case study also shows how concurrency easily adds exceptions of its own. Problems with adding or deleting observers during notification and preventing overlapping state changes are made more difficult by concurrency. Similarly asynchrony creates the need to cancel notifications, something that is usually not an issue in a sequential Observer implementation. This increase in sources of exceptions emphasizes the need for efficient concurrent exception handling mechanisms.

The case study reveals how concurrency makes it difficult to decide *where* and *when* to handle exceptions. This is something that is not caused by concurrency as such, but a sequential programming language often forces those decisions. If only one thread of execution exists, it is clear that that one thread must also handle all exceptions. Similarly exception handling semantics of the programming language makes it easiest to handle exceptions based on the current call-chain, since that is how the language finds exception handlers. If an exception is thrown in a sequential program, execution starts immediately handling that exception, not noticing other possible exception sources. Alternative approaches are possible in a sequential environment, too, but they require additional coding and reverting to mechanisms other than exceptions (coded return values, error flags, etc.). Concurrency and asynchrony simply reveal these existing problems.

Even though the exception mechanisms of this thesis do not help in *designing* appropriate exception handling strategies, the case study shows that they help in implementing them. For example, a generic component like the concurrent Observer needs to be able to react to exceptions unknown to it. Exception reduction allows Observer to reduce and choose from these exceptions when handling them. Reduction contexts make it possible for the actual application to pass reduction functions to the generic code, so that the generic code may perform reduction based on strategies chosen by the application.

Writing the case study proved that exception handling in generic code was difficult. Even in a simple component like the Observer several viable exception handling strategies can be found and different applications need different strategies. Only providing one strategy would make the Observer implementation only usable in applications where that one strategy was suitable. Passing application dependent reduction functions to the generic code solves part of the problem, but problems like “Where should exceptions be propagated to?” were still solved by choosing and fixing one strategy from many. In those problems exception reduction mechanisms did not provide any help.

6.3 Case study: An Application using Observer

To evaluate the implementation of the concurrent Observer pattern, a simple application using the pattern implementation was written. The Observer pattern is ideal in situations where several objects want to modify their state based on the

state changes in other objects. The objective of the evaluation application is to test the practicality of the Observer pattern implementation and exception reduction in a realistic application.

The source code of the case study application can be found in `http://www.cs.tut.fi/ohj/kcpp/observer-html/`.

6.3.1 Structure of the case study

Image manipulation is one area where observer pattern fits well. Especially in photography, *non-destructive* editing has become widespread. In non-destructive editing, several image manipulation steps are performed on a photograph. Settings for each step can be tweaked even after new manipulations steps have been added. If the parameters of one step are changed, the following manipulation steps are reapplied to the image. This allows the user to create a chain of image manipulation steps and then to adjust the parameters of the steps while all the time seeing the final result. Adobe Photoshop Lightroom [Adobe Systems, Inc., 2008] and Nikon Capture NX [Nikon Corporation, 2008] are examples of commercial photo editing software based on non-destructive editing.

Actual image manipulation in this case study is done using the Magick++ library [Friesenhahn, 2007], an open source C++ image manipulation library which internally uses the ImageMagick library [ImageMagick Studio LLC, 2008] (similarly open source). For its graphical user interface, the application uses FLTKC++ library [FLTK, 2008].

In this section, the structure of the case study application is described and its concurrency issues are discussed. Then found exception issues and multiple exception handling are covered, especially in terms of exception reduction.

The application was written so that it could be compiled both as a normal sequential program without KC++ and as a concurrent application using KC++. Because KC++ does not extend the C++ syntax, this required only minimal changes to the application. Performance of the sequential and concurrent versions was compared to get some indication on the overhead of the KC++ system compared with traditional sequential C++. Those results are given in Section 6.4.

6.3.2 Structure of the application

The structure of the case study application is simple. Each image editing step uses two active objects. A “settings object” contains a graphical user interface to edit the settings of a step. An “imager object” performs the actual image manipulation based on the settings. All communication between objects is asynchronous. This means that imager objects can perform their calculation concurrently, if they do not depend on the results of each other. Additionally calculation in imager objects is separated from settings objects, which means that the user interface stays responsive even when calculation is in progress. Figure 6.5 on the following page shows the basic structure of the application. Only two imager types are shown in the diagram. The diagram also shows the main program whose user interface is used to create and connect imagers by registering them as observers of each other.

An imager object acts as an observer of its settings object. This way image manipulation can be reapplied each time its settings change. In addition to this, imager objects can register themselves as observers of other imager objects. This allows stacking manipulation steps on top of each other. When an image changes, imager objects depending on it are notified, and they re-calculate their manipulation. Some imagers have no source imagers, like a "load imager" which gets its image from a file or a URL. Most imagers have one source imager, like a blur imager which calculates a blurred version of its source image. Finally, a combine imager has two source imagers and combines them to a single image using a specified algorithm and parameters.

Figure 6.6 on page 145 shows a screenshot of GuiImager in action. Each active object in the application has its own window. The windows of settings objects contain settings user can change, and the windows of imager objects show the resulting images. In addition to these, each window contains a message box for debugging. Imager windows also contain two indicators to show whether an imager is in the middle of a state change or waiting for the completion of notification (these indicators are updated by overriding virtual functions in the `Subject` class in the Observer implementation).

6.3.3 Concurrency in the application

This section discusses some concurrency issues in the implementation of the GuiImager application.

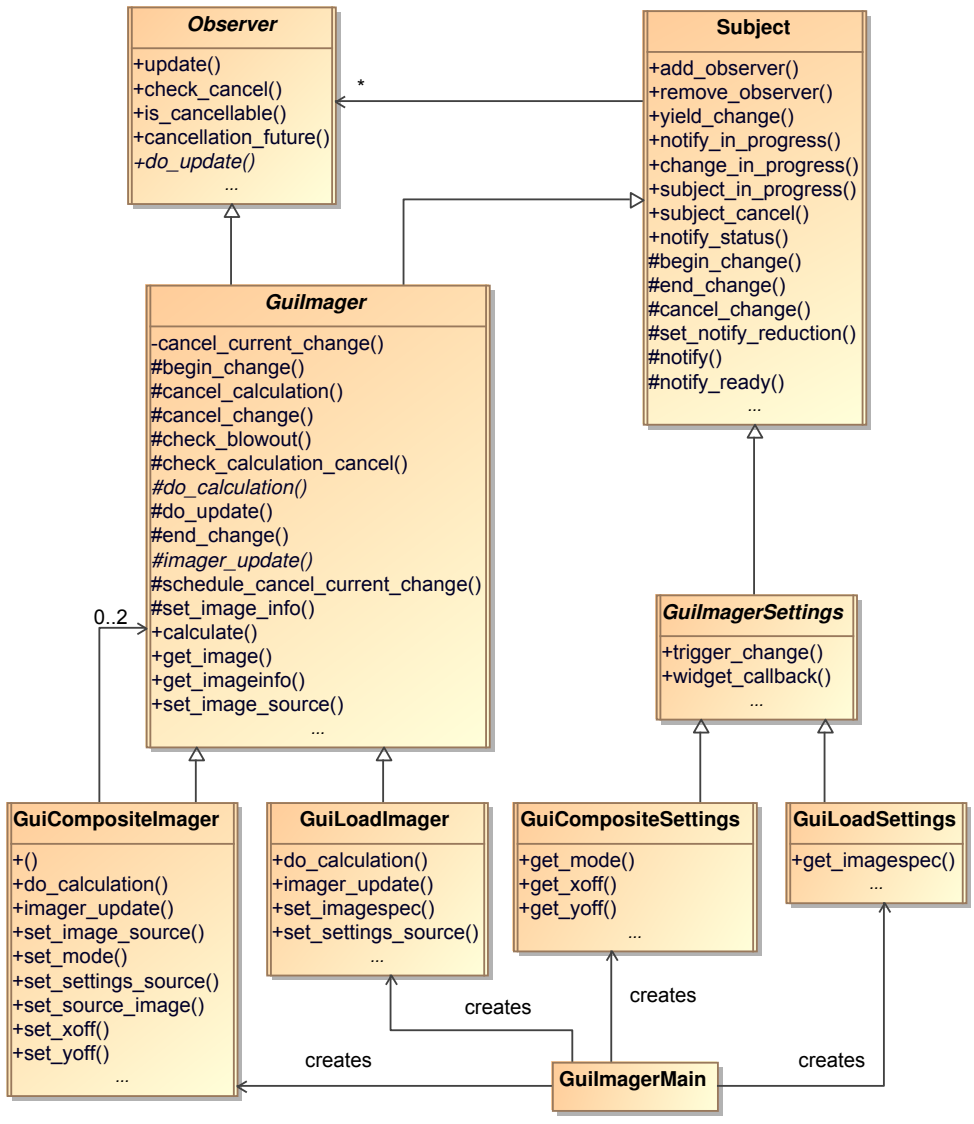

FIGURE 6.5: *Structure of the GuiImager case study*

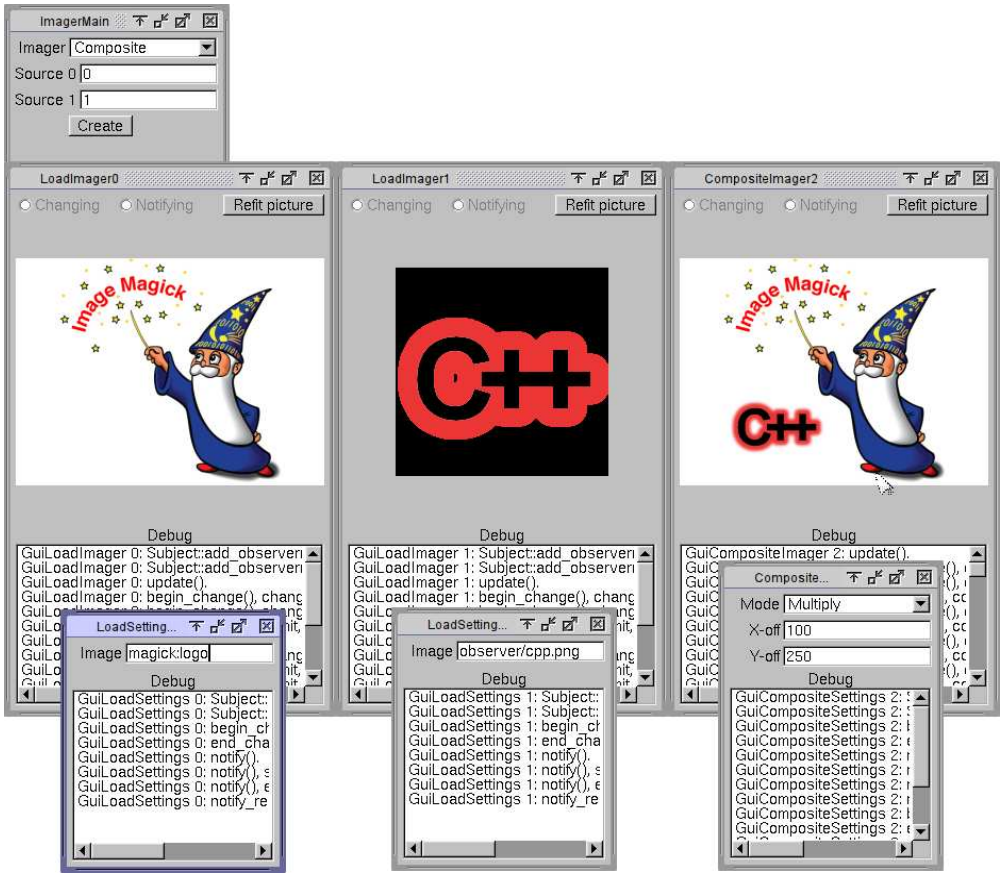

FIGURE 6.6: *Screenshot of GuiImager in action*

Active objects in GuiImager

The FLTK user interface library and the KC++ method parser both have an event loop which waits for messages and reacts to them. The application has one process for each active object, so combining these event loops was necessary. This was done by registering the KC++ method parser to the FLTK library as a callback function. Normally each process in the application waits in the FLTK event loop. If an active object message (a method call or a future value message) arrives, the FLTK event loop calls the KC++ method parser to handle the message. If a user interface event occurs, the FLTK event loop calls a callback function which in turn calls an appro-

priate method of the active object. The call is a normal C++ member function call, not an asynchronous KC++ method call. This solution allows the active object and the FLTK user interface for that object to share the same process.

The decision of running active objects and their user interface in the same process was arbitrary. It would have been just as simple to run the whole user interface in one process and limit active objects to image processing. However, as a concurrency evaluation case study the used approach was considered more appropriate. It makes active objects more concrete and allows easy measurement of the latency caused by running active object methods. Additionally, this approach allows different windows of the user interface to work concurrently. It also allows active objects to easily control their own windows (display debugging messages, change update and notification indicators etc.).

One result of sharing the same process is that the graphical user interface is frozen while its active object executes its methods. This is not a problem in this application, since all heavy computation is performed by imager objects which do not have any user inputs (their windows only contain a picture of the processed image and debugging indicators). When a user changes a setting in a setting window, this only triggers notification in the settings object. Since notification calls observer updates asynchronously, process execution returns to the user interface code almost immediately.

An example sequence

As an example of how GuiImager works, Figure 6.7 on the facing page shows a sequence diagram of a notification chain. This example contains two load imagers which load their image from a file. The resulting two images are combined using a composite imager.

At the beginning of the sequence, user enters a new file name to the load settings window of the first load imager. This triggers notification in the load settings object, which signals its load imager. The load imager queries the new file name, loads the image from the file and then enters its own notification phase. This signals the composite imager, which in turn queries both its load imagers as well as its settings object. All the queries are asynchronous and the results are stored in futures, so imagers begin their calculation methods without waiting for the results.

When calculation is complete, the composite imager does its own notification. When that is complete, the results are sent to the load imager. This completes the

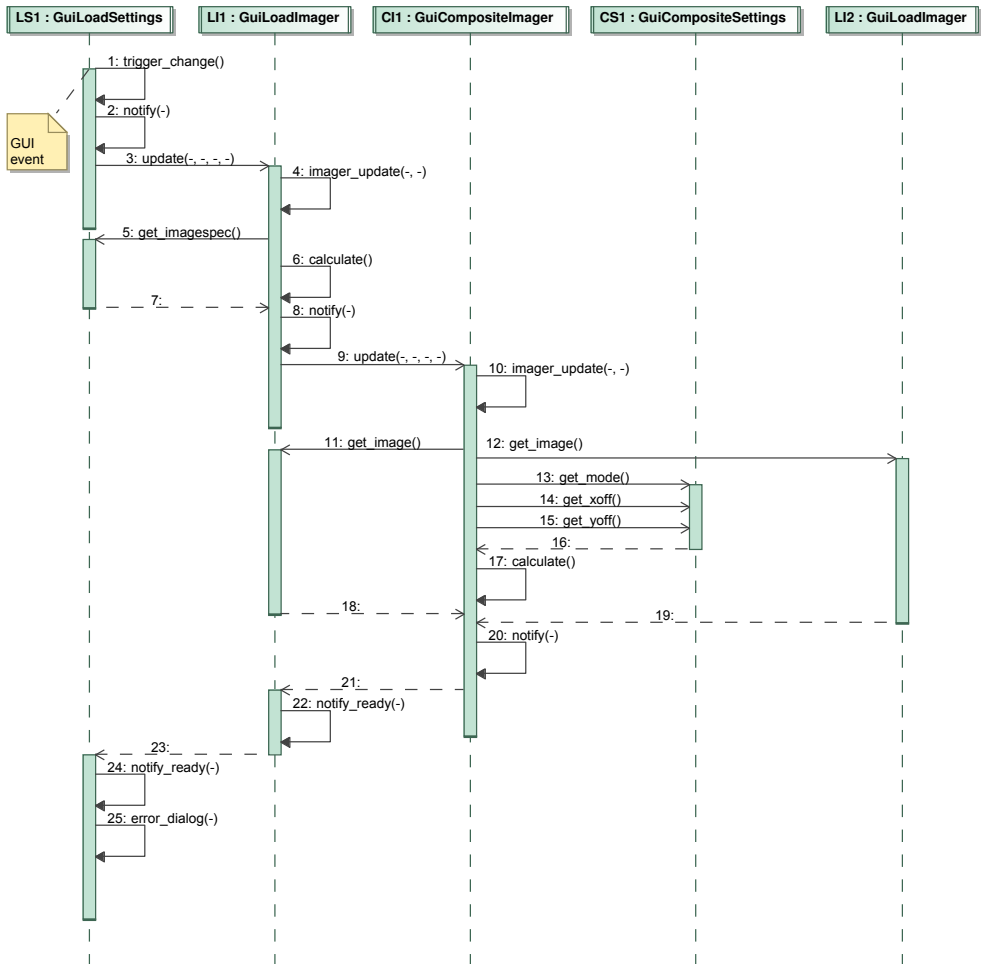

FIGURE 6.7: *Sequence diagram of a notification chain*

load imager notification and the results are sent to the load settings object, completing its notification. If the notification contained an exception (thrown somewhere on the way, and reduced), the load settings object shows an error dialog to the user.

Cancelling an ongoing update

The Observer implementation does not allow a subject to change its state while notification (or another state change) is in progress. This must be taken into account while designing imagers and their settings objects. The Observer implementation throws an exception if state change is attempted during notification or state change. The Observer implementation also provides methods for polling the status of the subject.

The state change attempt in both imager and settings objects occurs from inside the methods of the active object itself. The subject does not execute methods from other callers until the method has been completed. This makes it safe to first poll the subject to check that it is idle before starting the state change. If the subject is waiting for notification to finish, `subject_cancel()` is called to attempt cancellation of the notification. `subject_cancel()` returns a future which becomes ready when notification has ended. This future can be used to attempt the state change again later.

If state change was attempted from outside the subject, polling the subject status and then calling a state changing method is not safe. It is possible that another caller initiates a state change after polling but before state change is called. For this reason, the best strategy would be to simply call a state changing method. If an exception results, `subject_cancel()` can be called and a reattempt can be scheduled after cancellation is ready. The GuiImager application does not contain state changes of this kind, however.

6.3.4 Exception handling

Exceptions in the GuiImager application come from three sources: the Observer pattern implementation, the Magick++ library and the GuiImager application itself. Possible exceptions from the Observer pattern implementation have been covered in Section 6.2.4.

The Magick++ library has its own exception hierarchy which it uses to signal problems and errors in image manipulation. This hierarchy is a good example

of a third party exception hierarchy not designed to be used concurrently. The base of the Magick++ exception hierarchy is `Magick::Exception` which is derived from `std::exception`. Derived from this are two further bases `Magick::Warning` and `Magick::Error`. Warnings indicate non-fatal problems which may affect the quality or completeness of the result. Errors in turn signal fatal problems which prevent a meaningful result from an operation.

Magick++ exceptions are not derived from KC++ exception base classes, so they cannot be passed directly between active objects. KC++ counterparts for these exceptions were generated using the `ConcExcp` and `ConcTraits` templates (Section 4.5). This also makes it possible to use these exceptions in exception reduction. Listing 6.3 shows the definition of the KC++ counterpart for Magick++ exception `Magick::Warning`.

The KC++ exception mapping functionality automatically converts raw Magick++ exceptions to their KC++ counterparts, if a Magick++ exception escapes an active object

```
1  // Marshalling operators
2  OMsg& operator<<(OMsg& omsg, const Magick::Warning& excp);
3  IMsg& operator>>(IMsg& imsg, Magick::Warning& excp);
4
5  // Concurrency trait for RPC, exception mapping and folding
6  template<>
7  struct ConcTraits<Magick::Warning>
8  {
9     // Inheritance relationship for exception mapping
10    typedef Magick::Warning MapBases(Magick::Exception);
11
12    // Folding information, does not fold from this level up
13    typedef Magick::Warning FoldBases();
14
15    // Folding function concatenates error messages
16    static Magick::Warning* create_folded(CompoundException& ce)
17    {
18       std::string msg;
19       RpcExcpIterator<Magick::Warning> b=ce.begin<Magick::Warning>();
20       RpcExcpIterator<Magick::Warning> e=ce.end<Magick::Warning>();
21       for (RpcExcpIterator<Magick::Warning> i=b; i!=e; ++i)
22       {
23          msg += (*i)->what();
24          msg += '\n';
25       }
26       return new Magick::Warning(msg);
27    }
28 };
29 // Actual declaration of exception, registers exception mapping
30 typedef ConcExcp<Magick::Warning> ConcMagickWarning;
```

LISTING 6.3: *Declaration of Magick++ exception counterpart*

method. As described in Section 4.6, the mapping can slice the exception object if a proper KC++ counterpart exception has not been declared. However, since KC++ counterparts are defined for all exceptions used by the GuiImager application itself, possible slicing does not affect exception handling in the application.

Some additional exceptions were added to the application to make exception reduction more interesting. These exceptions are all derived from `ImagerException`:

- `BlownOutImager` is thrown if an imager operation results in "blown highlights", i.e. some pixels get values which do not fit into the data type used by the application.
- `InvalidImagerSettings` results if the user inputs settings which are invalid for the operation. GuiImager uses string input fields for all settings in its user interface to make it easier to create exceptions of this type.
- `EmptyImager` exception means that an operation cannot be carried out because a source imager is empty. An imager can be empty if it has been just created or if its previous image manipulation operation failed.
- `CalculationCancelled` is thrown if an imaging operation is cancelled. This exception is derived from `ObserverCancellation` of the Observer pattern implementation.

Some exceptions affect the application in ways that require notifying the user. In GuiImager, the user interface callback function triggers a state change in the settings objects, and thus initiates Observer notification. The eventual result of the notification is returned in a future. To notify the user if the future contains an exception, a method is scheduled to be executed when the future becomes ready. This method shows the user a dialog window if the future contains an exception.

Exception reduction

Exception reduction is needed in the GuiImager application in places where multiple asynchronous calls create a need to reduce multiple exceptions. The goal of the reduction in GuiImager application is to choose, combine and reduce exceptions so that a suitable error message can be shown to the user.

There are three places in the GuiImager application where multiple exceptions may occur: when a settings object notifies several imager objects, when an imager

object notifies several other imager objects, and when an imager updates its state and queries its settings object and source imagers (queries are done asynchronously, so multiple exceptions may result).

Exception reduction is needed to calculate the final exception in all of these cases. In the first two cases reduction happens inside the Observer implementation, as in Section 6.2.7. Settings and imager objects just have to provide a suitable reduction context to the Observer implementation. The last case requires exception reduction in the actual GuiImager code.

Because the GuiImager application consists of several notification chains, all concurrently occurring exceptions are not necessarily handled in a single reduction step. Each notification and imager calculation performs reduction, but exceptions reaching that point may already be results of earlier reductions. Since each imager may potentially introduce its own exception classes, it is possible that the set of exceptions to be reduced contains exceptions that are unknown to the reduction context.

Designing reduction strategies proved challenging even for such a small application. The goal of reduction in this case is to provide a meaningful error message for the user. If several exceptions occur as the result of user's actions, it proved difficult (at least to the writer of this thesis) to even *define* what kind of error message would be appropriate in each case.

This was especially the case if several different types of exceptions are thrown. Traditional sequential programs automatically choose the first exception they encounter, and do not become aware of the possibly more fatal exceptions that would have occurred later. Futures, asynchronous calls, and exception reduction make it possible to react to several exceptions, but they do not help in defining what should be done in such a case. If different exceptions have a mutual cause, they can be reduced to a single exception. However, if there are several causes of exceptions, it is difficult (or at least was difficult in GuiImager) to define the "best" strategy for reducing the number of exceptions.

Implementation of reduction

In the end, reduction was based on the following rules. These rules are listed in the order of importance (which in many cases is a matter of opinion):

- Errors from Magick++ are kept intact. If several such errors exist, they are folded into one.
- All application dependent GuiImager exceptions can be folded together.
- Warnings from Magick++ can be folded into a single exception. In folding, the warning messages in exception objects are concatenated together.
- If calculation is cancelled by an interrupting operation, all such exceptions can be folded together.
- If operations fail because source imagers are in the middle of their calculations, these exceptions are removed. These exceptions are not important, since when the source imager completes its calculations, it triggers notification, updating the observer.

The list above mentions exception folding (Section 5.7) in several places. In this application folding was done using `KC++ FoldAnyUpTo` reduction class. This reduction selects all exceptions derived from the specified base, finds their most derived common base, and replaces the exceptions with a single exception of that base. The new exception is created using the folding constructor or function of the class. For example, all Magick++ can be folded together using reduction class `FoldAnyUpto<Magick::Warning>`.

Listing 6.4 on the facing page shows the reduction class used in this case study. The topmost `ModifyAndReduce` combinator is used to modify the set of exceptions before actual reduction. In this case, `ReplaceCompounds` is used to "flatten" the exceptions by replacing compound exceptions with their contents. The actual reduction is performed on the result.

The second part of the reduction is choosing and folding exceptions based on the list shown earlier. That is done using `ChooseAndThrown` combinator. That reduction combinator tries each reduction class in sequence. When it finds one that accepts the set of exceptions (or a part of it), it uses that reduction class as primary reduction which provides the result of reduction. After the primary reduction, thrown reduction of the rest of the reduction classes is still attempted in order to further reduce the set of remaining exceptions.

Exception reduction only works on exceptions that are passed through futures from other active objects. It cannot control normal `C++` exceptions thrown locally (as discussed in Section 5.5.4). This limitation shows clearly in GuiImager application.

```
1  typedef CombineRC<
2    Reduc<
3      ModifyAndReduce(
4        ReplaceCompounds,
5        Reduc<ChooseAndThrown(
6          FoldAnyUpto<Magick::Error>,
7          FoldAnyUpto<ImagerException>,
8          FoldAnyUpto<Magick::Warning>,
9          FoldAnyUpto<CalculationCancelled>,
10         Remove<InProgress>,
11         ThrowOneOrCompound
12       )>
13     )>
14   >
15   CalculationReductionClass;
```

LISTING 6.4: *Definition of the reduction class used in GuiImager*

Image manipulation in each imager consists of querying imager settings and source images using asynchronous calls, and performing the actual calculation. Exceptions in the queries are RPC exceptions passed through futures, whereas the calculation may result in an exception that is thrown normally from the Magick++ library performing the calculation.

Ideally, exception reduction should be performed on all the exceptions mentioned above. If the program encounters an exception in a future, exception reduction can be performed by a future group, and the result of the reduction can be thrown. However, if the first exception comes from the Magick++ library, then that exception is already thrown and reduction can be performed only after that exception is caught and added to the set of exceptions to be reduced. Catching already thrown normal exceptions requires extra code in the application, but no suitable solution has been found to prevent this.

Listing 6.5 on the next page shows calculation code in one imager class. Actual calculation is performed in a try block, using a reduction context and a future group to manage multiple exceptions resulting from asynchronous calls. The try block is followed by three catch blocks.

The first catch block catches all KC++ exceptions. This catch block is entered in two situations. Either the future group triggered exception reduction, which results in a KC++ exception, or a KC++ exception was thrown outside the control of the future group. The code in the catch block checks the status of the reduction context to distinguish the two cases. In the first case, exception reduction has already been

```
1  Future<void> GuiLevelsImager::calculate()
2  {
3     ChangeScope sc(changescope());
4     ReductionContextRC<CalculationReductionClass> rc;
5     try
6     {
7        FutureGroup fg(rc);
8        fg.add(source_);
9        fg.add(blackpoint_);
10       fg.add(whitepoint_);
11       fg.add(gamma_);
12
13       // The performance tests return here
14       if (!usemagick()) { return sc.commit(); }
15
16       Magick::Image im = source_;
17       double bp = blackpoint_/100.0*(QuantumRange);
18       double wp = whitepoint_/100.0*(QuantumRange);
19       im.level(bp, wp, gamma_);
20       image() = im;
21       // Check for possible blown out highlights
22       check_blowout(0, 0, image().columns(), image().rows());
23    }
24    catch (const RpcExcpBase& e)
25    { // We are here either after reduction of because of out-of-fg exception
26       if (!rc.reduced())
27       { // No reduction -> out-of-fg exception
28          rc.addException(e);
29          rc.reduce();
30       }
31       else { throw; } // Pass through already reduced exception
32    }
33    catch (const std::exception& e)
34    { // Create a concurrent exception and add to the reduction context
35       if (RpcExcpBase::SharedEP me = RpcExcpBase::createMapped(e))
36       {
37          rc.addException(me);
38          rc.reduce(); // Reduce and throw
39       }
40       else
41       {
42          assert(!"std::exception which could not be mapped!");
43       }
44    }
45    catch (...)
46    {
47       assert(!"Unknown exception!");
48    }
49
50    return sc.commit();
51 }
```

LISTING 6.5: *Calculation in levels imager*

performed, so the chosen exception can be propagated further by re-throwing it. In the second case, reduction was not performed. In this case the code in the catch block inserts the thrown exception to the reduction context and explicitly calls reduction.

The second catch block is for situations where a non-KC++ exception was thrown, but the exception was still derived from `std::exception`. In this case reduction has not been performed since it always results in KC++ exceptions. The code attempts to map the thrown exception to a KC++ counterpart using exception mapping (Section 4.6). This is possible if a suitable mapping has been declared using `ConcTraits`. If the mapping succeeds, the resulting KC++ exception is inserted into the reduction context and reduction is performed.

Otherwise the non-KC++ exception cannot be mapped to a KC++ exception, meaning that it cannot be propagated out of the active object. In this case, the program is terminated with an error message. This catch block shows how non-KC++ exceptions can be generically mapped to KC++ counterparts for reduction. Without this catch block the non-KC++ exception would propagate into the KC++ method parser, which would also attempt mapping. However, in this case exception reduction would not be performed.

Finally, the third catch is for catching exceptions which are not KC++ exceptions nor derived from `std::exception`. These should not occur, so the catch block simply terminates the program with a failed assertion.

6.3.5 Discussion on concurrent exception handling

The case study shows that concurrent exception handling mechanisms presented in this thesis can be used in practise. It also revealed that *designing* concurrent exception handling was harder than expected. It was difficult (at least to the author) to choose the best strategy in the presence of multiple exceptions. In many cases there were several possible alternatives and no clear reason to choose any one of them. However, when an exception reduction strategy had been chosen, it was straightforward to implement it using ready-made reduction functions, reduction combinators, and exception folding.

Since the case study used a third-party library, Magick++, it was a good test case for the usability of the RPC exception passing mechanism in the presence of third-

party exceptions. The case study shows that the mechanism is usable and necessary additional code (serialisation and exception mapping) is straightforward to write. Application code can continue to use original third-party exceptions in most of its code, except in exception reduction, where RPC-enabled exception classes must be used.

Use of the Magick++ library also demonstrates why exception folding requires additional mapping information and why folding based on inheritance hierarchies requires limitations. The Magick++ exception hierarchy is divided into two top-level branches, errors and warnings. Even if several Magick++ exceptions are reduced and folded together based on the hierarchy, it is essential that information on these two categories is not lost. If exception folding would simply reduce exceptions to their most derived common base class, then reducing a Magick++ warning and error would result in a top-level Magick++ exception object, without information on whether a warning or a fatal error had happened. Concurrency traits presented in this thesis make it possible to limit folding to these two categories so that errors can be folded together as well as warnings, but mixing of these categories never occurs.

Use of a third-party library also clearly shows how limitations in the C++ exception handling make it difficult to group together RPC exceptions from futures and normal exceptions thrown by the library. Handling of normal C++ exceptions is unaware of RPC exceptions and exception reduction, so extra code is needed to catch, map and combine those C++ exceptions with RPC ones. This extra code degrades readability of the code and makes writing concurrent exception handling unnecessarily complex. This is clearly a shortcoming in the mechanisms in this thesis, but no solution has been found, due to inflexibility of the C++ language.

This case study revealed an interesting situation in concurrent exception handling. The components in the application form a graph of concurrent subjects and observers, and there are several places where multiple exceptions arise. This means that exceptions go through several reduction steps, where the results of earlier reduction participate in later reduction. In the case study this did not cause problems, but more generally it can make writing exception reduction more difficult, since reduction steps must work together nicely in order to produce acceptable results. To lesser extent similar challenge already exists in sequential exception handling, where it is possible that exception handlers rethrow exceptions or throw new ones. This results in a chain of exception handlers, which must also work together nicely.

For some cases exception reduction is a good solution based on the case study. Exception folding can be used in places where it is acceptable to abstract away specifics of exceptions in order to reduce them to a single more abstract exception. For other cases, exceptions or exception categories can be prioritised and used for reduction.

However, the GuiImager application clearly shows some problems with multiple exceptions. An exception class hierarchy forms a hierarchy of exception categories, but as such tells nothing about the importance of exceptions in relation to other exceptions. There are situations where exception reduction is complicated or even unclear. For example, the reduction currently used in the case study says that among GuiImager exceptions, in-progress exceptions are the most important, followed by cancellation exceptions and then the rest of imager exceptions.

However, even in this simple case study imager classes are allowed to derive exceptions of their own, and those exceptions may be important to the user. Therefore reduction should not choose an in-progress exception as the result of reduction if there are other unknown GuiImager exceptions in the set of exceptions to be reduced. Nor should the reduction abstract away those important exceptions by combining them with the rest of GuiImager exceptions.

It can be argued that the reduction should only reduce types of exceptions that "are known" at the point of reduction. It is possible to write a reduction function having that knowledge, but including a list of all known exception types in the reduction is tedious regardless of whether such reduction was written by hand or by using some sort of reduction combinator. Such reduction would also be difficult to maintain because the list of known reductions should be updated if new exceptions are added to the application.

The case study also clearly shows one limitation with future guards. Currently they wait for all registered futures to become ready before performing exception reduction. On one hand this is reasonable, because full exception analysis is only possible after all exceptions have been received. On the other hand, it causes a delay in exception handling. In some cases it would be useful to proceed with exception reduction without waiting further, if a fatal enough exception has been received. This would become possible if the programmer could provide a function, which would decide whether to proceed with exception reduction or continue waiting. However, this would complicate exception handling even more.

6.4 Performance evaluation of Observer and GuiImager

The performance measurements given in Section 6.1 are one way to measure the performance impact of concurrent exception handling. However, such minimal tests do not include the inevitable overhead present in a real-world application. To get an idea of how concurrent exception passing and reduction affects performance in a real application, performance of the GuiImager application and its exception handling was evaluated.

6.4.1 Test setup

The test setup was simple. The test creates a load imager to load an image from a file and a modulation imager to change its brightness and saturation. Then the image is repeatedly changed to another image. When exception handling is tested, the brightness and saturation values are left empty. This causes two exceptions to be created, reduced and propagated through the application. Figure 6.8 on the facing page shows a sequence diagram of one test cycle resulting in exceptions. The figure also shows the places where exception reduction takes place during the test.

The speed of the GuiImager application is affected by several factors. The cost of exception handling and exception reduction is the factor this thesis is interested in. However, performance of the application also depends heavily on the speed of the user interface library and the time it takes to perform actual image processing operations. Especially the image processing part is problematic since its effect can be made arbitrarily small or large by increasing or decreasing the size of images to be processed. Similarly the time used in the user interface depends on the performance of the user interface library, the graphics subsystem of the operating system, the speed of the graphics card, etc.

For these reasons, actual image processing was left out from performance tests. A flag was added to the program, and Magick++ image processing functions were not called if the flag was set. Another flag was added to cause the program to skip calls to the FLTK user interface library. In the test setup, the imager active objects were triggered from a loop in the main routine, so a graphical user interface was not necessary to run the tests.

A “conventional” C++ version running GuiImager was needed to see the impact of KC++ exception handling mechanism. To achieve this, a sequential GuiImager application was compiled without using KC++. This was quite straightforward since

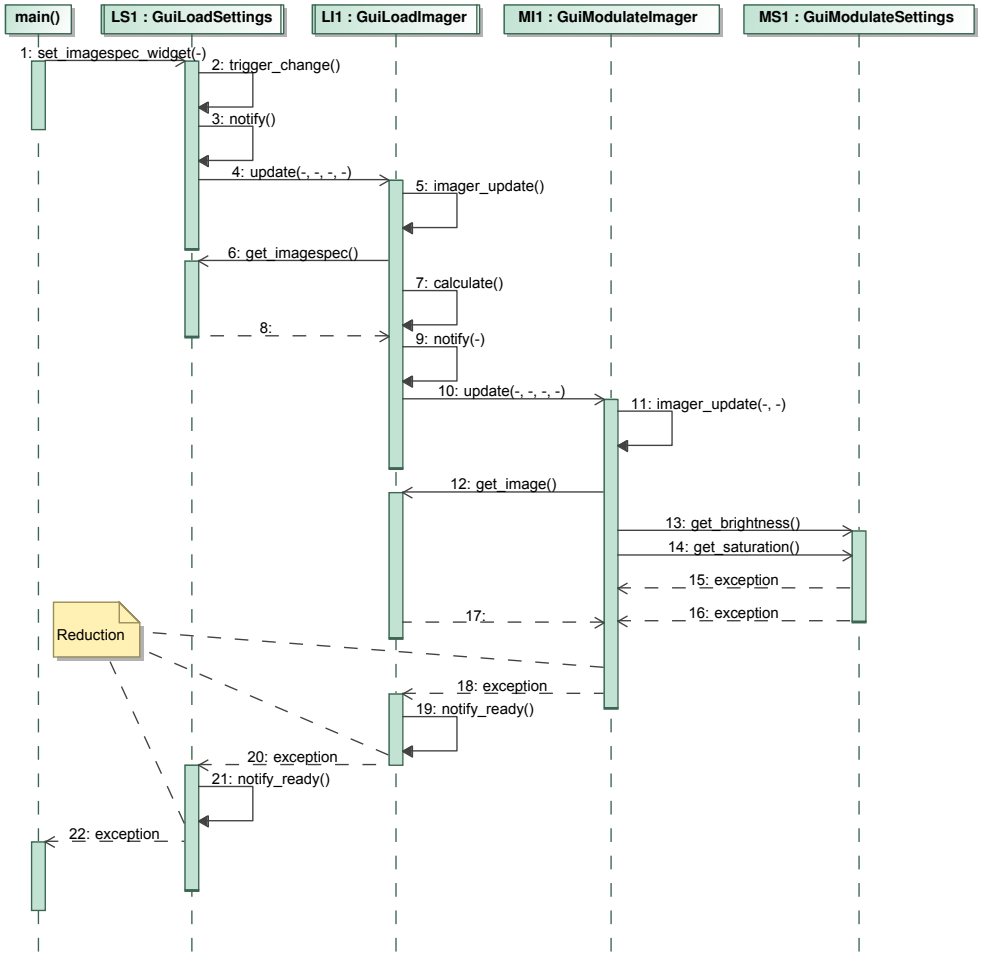

FIGURE 6.8: *Sequence diagram of a test cycle resulting in exceptions*

KC++ does not change C++ syntax. The non-KC++ version just required dummy versions of KC++ futures and the base class `Active`. The dummy future classes were written completely using inline functions so that the compiler could optimise away futures, i.e. a `Future<int>` return value was optimised to a plain `int`.

The following test setups were used:

- A sequential non-KC++ version as a baseline, both with and without throwing exceptions. This version used no graphical user interface or image processing.
- A KC++ version with no graphical user interface or image processing, both with and without throwing exceptions.
- A KC++ version with no graphical user interface or image processing and exception reduction disabled to see the cost of reduction.

6.4.2 Test results and analysis

All tests were run in the same circumstances as the earlier tests (OpenSuse 11.0 64-bit Linux with kernel 2.6.25.18-0.2 and Intel Core 2 processor running at 2380 MHz, CPU frequency scaling off). Only one CPU was enabled in the tests to prevent hardware parallelism from affecting the tests. Again GCC 4.3.2 compiler was tested with both -O2 and -O3 optimisations, but both produced almost identical results. The *K-best* method [Bryant and O'Hallaron, 2003, Ch. 9] was again used to verify that results are representative ($N = 20$, $K = 3$ and $\epsilon = 0.01$).

The results of tests are shown in Table 6.2 on the next page and in graphical form in Figure 6.9 on the facing page. The first column in the table identifies the test. The second column shows the number of user interface events generated for the test, and the third column contains the time the test took to run. The fourth column contains the actual result, the time to process one event. These figures were rounded to two significant digits because of the selected K-best parameters. The last column in the table shows how much the fastest test and slowest test results differ from each other. This gives an idea how much fluctuation the test environment caused in the test.

Normal C++ exception handling

Exception handling in sequential tests 1. and 2. consists of an empty catch handler in the main routine. When the first exception occurs (brightness in the modulate settings object has not been set), program execution is transferred to the main routine.

	Test	# of cycles	Time (s)	Time/cycle (µs)	Δ_{max} (µs)
1.	Normal C++, no exceptions	10^7	81.33	*8.1*	0.40
2.	Normal C++, exceptions	10^6	88.98	*89*	2.2
3.	KC++, no exceptions	10^5	46.59	*470*	4.7
4.	KC++, exceptions	10^5	68.02	*680*	6.6
5.	KC++, no reduction, no exceptions	10^5	45.87	*460*	5.7
6.	KC++, no reduction, exceptions	10^5	67.49	*670*	4.3

TABLE 6.2: *Results of GuiImager performance tests*

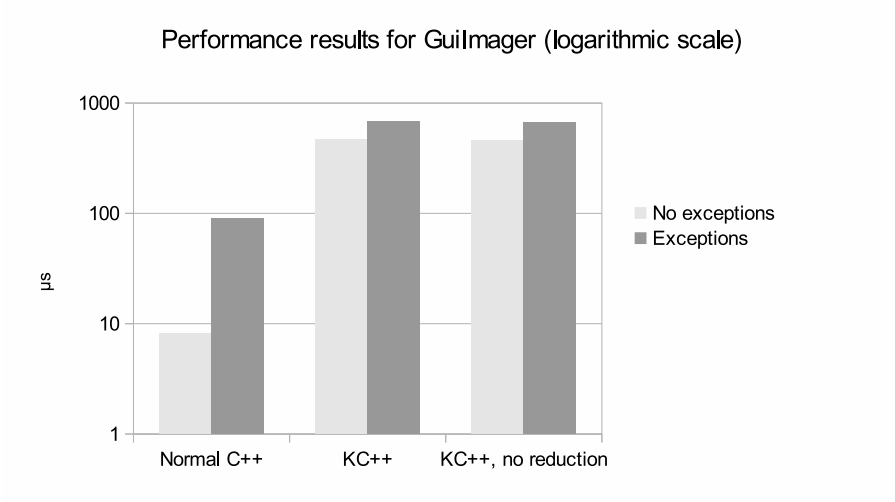

FIGURE 6.9: *Test results in graphical form*

No exception reduction or other exception handling mechanisms described in this thesis are used.

The overhead of exception handling still dominates, causing the program to run over 10 times slower when an exception occurs. Since there is no actual exception handling code, the overhead is caused by find the appropriate catch block, and stack unwinding. The exception passing overhead is 81 µs. In the simple function call test in Table 6.1 on page 120, the exception passing overhead was only 4.8 µs. This 17 times difference shows that exception passing overhead in normal C++ can vary greatly depending on the application. The difference can be explained by stack unwinding. In the simple test, the exception was passed up one function call in the call chain. In the GuiImager test, the exception is passed up 19 levels. This causes

the stack unwinding to take much longer since each stack frame must be analysed for possible destructors.

KC++ exception handling

The concurrent and parallel versions of GuiImager trigger two exceptions in the modulate settings object, one for missing brightness value and one for missing saturation. These exceptions are propagated to the modulate imager object, which reduces them. The reduced exception is propagated to the load imager object, reduced again, propagated to the load settings object, reduced once more, and finally sent to the main routine. Each of these objects is active, so exceptions are passed between objects using the KC++RPC exception passing mechanism. A total of seven exception objects are thrown in the active objects during exception passing and reduction.

Lines 3. and 4. in Table 6.2 show that exception handling takes 1.5 times the time of normal program execution. This is reasonable compared with the earlier 3.5 times slower in the simple test (Table 6.1), since actual application logic is much more complex.

In absolute time exception handling overhead in GuiImager test was 210 μs when it was 24 μs in the simple test. This is caused by seven instead of two exceptions, exception reduction and stack unwinding. The ratio of 9 is somewhat similar to the ratio 17 in normal C++ case earlier.

When the KC++ version is compared with the normal C++ version, one interesting result can be noticed. The KC++ version throws 7 individual exceptions during exception handling, where normal C++ version throws one. Nevertheless, the overhead in the KC++ version is only 2.7 times larger. This can be explained again by stack unwinding. Even though 7 individual exceptions are thrown in 5 separate execution threads, each exception has to travel through a much shallower call chain to be caught. The combined depth of the call chains is not substantially larger than the single call chain in the normal C++ case. The KC++ version receives two exceptions from the modulate settings object instead of one, and active method invokers add to the call chain, but otherwise the exception travels the same route as in the sequential case. Since C++ exception handling overhead is dominated by stack unwinding, the number of exceptions thrown does not affect the performance much.

The previous result shows that exception handing between active objects does not suffer much from having to throw the same exception again as it travels from an active object to another. Exception handling performance is dominated by the

length of the path the exception travels, regardless of how that path is split between active objects.

Effect of exception reduction

Lines 5. and 6. in Table 6.2 show the results of tests when exception reduction code was commented out. In these tests multiple exceptions were handled simply by embedding them in a compound exception.

The results would seem to indicate that test cycles were 10 μs faster without reduction regardless of whether exceptions occurred or not. This is somewhat surprising since reduction code is executed only if exceptions occur. A possible explanation for this anomaly is that commenting out reduction code reduced the size of the executable enough to have a small impact on cache efficiency.

In any case the difference between tests with and without reduction is only 2 %, and not far from the normal maximum fluctuation between tests (shown in the last column). This indicates that the simple reduction functions used in this test application do not have significant impact on exception handling. The situation would of course change if the number of reduced exceptions and the complexity of reduction functions were increased.

Chapter 7

Related work

This chapter presents work related to themes in this thesis, especially exception serialisation in RPC and handling of multiple concurrent exceptions. Relevant parts of the new C++11 standard are also discussed here, since the work in this thesis has been done based on the current 1998/2003 version of the C++ standard.

7.1 RPC exception passing in C++

C++ needs special mechanisms for RPC exception propagation because RPC and serialisation are not part of the standard language and because the language does not force exception classes to be derived from a single base class. In contrast, in Java all exception classes must be derived from `Exception`, which in turn implements the `Serializable` interface. This makes it possible to pass all Java exceptions via RPC.

Several serialisation implementations exist for C++ and other languages. Many of these libraries are clearly written with object persistence in mind. This means that they are designed for serialisation and restoration of *values*, which is common for both persistence and RPC. However, this means that these libraries do not specifically handle propagation of exceptions. For example, many serialisation libraries are completely separate from method/function invocation mechanisms.

The idea in this thesis to provide a virtual `throwSelf` function to avoid limitations in static typing in the **`throw`** statement is logical. So it is no surprise that it is used in many places, e.g., it is part of the CORBA C++ Language Mapping Specification [OMG, 2003, §11.9] and the Qt framework [Qt Development Frameworks, 2011]. It

is also described by Herb Sutter and Jim Hyslop in their C/C++ User Journal column [Sutter and Hyslop, 2005], and by Andrei Alexandrescu in [Alexandrescu, 2001]. The Qt framework also uses a virtual `clone` method to replicate exceptions.

The idea to automate `throwSelf` and `clone` using CRTP was the author's own, but it turned out that the idea is used elsewhere too, at least to automatically provide dynamic cloning. The technique has been added to the Wikipedia article discussing CRTP pattern in 2010. [Wikipedia, 2011]

The implementation of the exception factory described in this thesis resembles Alexandrescu's object factories in [Alexandrescu, 2001]. However, Alexandrescu's factories are still based on user-provided creation functions. The idea to use the constructor of a static data member for registration of factories is loosely based on James Coplien's exemplars [Coplien, 1992]. However, this thesis automates the factories using inheritance and template metaprogramming.

While many object-oriented programming languages like C++ and Java support exception hierarchies through inheritance, CORBA does not support hierarchical exceptions in its IDL interface specifications. All exceptions are divided into user exceptions and system exceptions, and CORBA C++ mapping defines that all user exceptions are directly inherited from a provided abstract base class, making the hierarchy only one level deep. [OMG, 2004, §3.12] [OMG, 2003, §1.19]. Serialisation used in CORBA relies on user-written factory functions for creating objects during unmarshalling. For user-defined classes and types, it is the responsibility of the application programmer to write such factory classes, instantiate a factory from each, and then register those factories with CORBA run-time [OMG, 2003, §1.17.10].

Compared with KC++, the approach used in CORBA prevents use on third party exception classes, since all CORBA exceptions have to be derived from a CORBA exception base class. The writer of an exception class must also herself take care of writing the `throwSelf` method (called `_raise` in CORBA).

Separate serialisation libraries exist for C++, such as Boost Serialization library [Ramey, 2004], which contains support for versioned serialisation and restoration. The Boost Serialization library concentrates on serialisation of arbitrary values, so exceptions are not specially addressed in that library. It relies on several explicit mechanisms for dynamic factory creation, but does not require a common base class for serialised objects.

Similar serialisation of values using explicit dynamic factories is provided by the GNU Common C++ and its TypeManager class [Free Software Foundation, Inc., 2004].

Yet another serialisation library is s11n [s11n, 2005]. It uses the same virtual function based marshalling mechanism as the mechanisms in this thesis, but relies on explicit creation of dynamic factories.

Since these serialisation libraries are meant for general explicit serialisation and not exception handling, they do not have to rely on a common base class for serialisable classes nor provide support for throwing unmarshalled objects.

Although C++11 provides support for concurrency, it is limited to threads with a shared address space. RPC across address spaces is not supported, so serialisation is not provided either.

7.2 Exceptions and futures in C++

Exception handling mechanisms in this thesis are heavily based on futures for asynchronous communication and synchronisation. The current C++ language itself has no support for futures or concurrency, so such support must be provided. This thesis uses KC++ to give concurrency and futures. However, there are other C++ based concurrency libraries and platforms which use futures. The C++11 standard also has its own implementation of futures and concurrency.

Perhaps of the greatest importance is the concurrency support in C++11. C++11 contains library support for concurrent threads and futures. However, support for RPC calls across address spaces is not supported. In C++11, threads can be created as objects which execute a given function asynchronously. Futures are used for asynchronous return values. C++11 contains two kinds of futures, *unique futures* (`future`) and *shared futures* (`shared_future`). Their difference is that unique futures cannot be duplicated and so the asynchronous value is always referred to by only one unique future (unique futures can be *moved*, though, in which case the original future is cleared during copying). Shared futures behave like KC++ futures and can be copied and assigned freely, causing futures to share their value.

The interface of C++11 futures is quite limited. The value of a future can be asked, blocking the caller if the value is not yet available. Similarly it is possible to perform just synchronisation without accessing the value of the future. However, it is not possible to directly query the readyness of a future, but it is possible to give a timeout to a wait method, which can be used to get the same effect.

C++11 futures contain either a value or an exception, and a contained exception is automatically thrown if the value of the future is accessed. There are no queries to

find out whether a ready future contains a value or an exception, and the exception cannot be accessed without causing it to be thrown. These limitations make exception handling more difficult than with KC++ futures. One interesting feature of C++11 unique futures is that the value of the future can only be accessed once, and accessing the value moves the value out of the future, marking it invalid. This means that an exception stored in a unique future can only be thrown once, solving the problem of multiply thrown exceptions mentioned in Section 5.2.2.

The C++11 library also contains promises which are similar to KC++ future sources. They can be used to create futures and later pass values to them. Promises can also be used to pass exceptions to futures.

The Boost Thread library [Williams, 2008] contains support for concurrency and futures for C++. The C++11 threads and futures are based on the Boost library, but there are some differences between the two. Boost also contains both unique and shared futures. The interface of futures is somewhat larger in Boost and contains member functions for querying whether a future contains a normal value or an exception (like KC++ futures). This way it is possible to query about the existence of an exception without causing it to be thrown. The Boost Thread library also provides promises which are almost identical to C++11 promises.

The Qt framework [Qt Development Frameworks, 2011] provides concurrency support in C++ using some mechanisms similar to those in this thesis. Qt provides asynchronous return values using futures (called `QFutures`). It also requires concurrent exceptions to be derived from a common base class `QtConcurrent::Exception`. The Qt concurrency only supports threads in the same address space, so exception serialisation is not needed.

Concurrency can also be added to C++ by extending the language. The C++// language [Caromel *et al.*, 1996, Baude *et al.*, 1996] is in many ways similar to KC++. C++// also uses the existing C++ syntax and uses inheritance to mark classes as active. The C++// compiler converts C++// code to normal C++ code, and uses *wait-by-necessities* (a form of future) to synchronise with asynchronous method calls. The wait-by-necessities are not parametrised in C++//, but rather they are created using inheritance. This means that wait-by-necessities cannot be created for built-in types. The C++// does not support exceptions.

μC++ [Buhr *et al.*, 1992] is a concurrent extension of C++ which takes a different approach with exceptions. μC++ also has futures, and exceptions can be stored in these and propagated when the value of a future is requested. However, this only

happens when a server explicitly stores an exception in a future. If an asynchronous call raises an exception which it does not handle itself, that exception is automatically propagated to the caller and raised there, causing exception handling in the caller. The effect of μC++ exception handling model on concurrent exceptions is discussed in Section 7.3.

7.3 Multiple concurrent exceptions

This section discusses how the combination of exception handling and concurrency has been implemented in several languages and systems, especially in case of multiple concurrent exceptions.

A general analysis of concurrent exception handling in object-oriented systems can be found in [Romanovsky and Kienzle, 2001]. Ideas for handling multiple concurrent exceptions in a distributed system have been developed in [Xu *et al.*, 2000]. This work also introduces the idea of exception resolution. The idea of adding exception resolution to Ada and Java has been analysed in [Romanovsky, 2000].

7.3.1 Keeping exceptions thread-local

The simplest way to get rid of problems with multiple exceptions is to keep exceptions local to a thread of execution. This way two concurrently raised exceptions cannot simultaneously end up in one thread. If this strategy is chosen, it is still necessary to define what happens if an exception is raised but not handled within a thread. Two typical choices are either to terminate the thread, or ignore the uncaught exception.

The strategy used in the Ada language [Ada, 1995] is quite common in other languages, too. In Ada, new tasks (threads) can be created and the new tasks start executing asynchronously with their creator. Problems with multiple concurrent exceptions have been avoided by declaring that if an exception tries to leave the task body (i.e. if the task does not handle the exception locally), the exception is not propagated further and the task in question is terminated.

Ada-95 provides also *asynchronous remote procedure calls.* Since remote calls are procedure calls, they cannot have return values. If an asynchronous remote procedure call results in an exception after the caller has continued its execution, the exception is simply ignored.

The Java language [Arnold and Gosling, 1998] also has built-in concurrency. The exception handling strategy in the language is very close to the Ada approach. All exceptions are handled locally inside a thread, and they are lost if the thread does not contain an appropriate exception handler.

Although the exception mechanism in Java is close to Ada, different concurrency features affect exception handling in both languages. These differences (as well as differences in other concurrency features) are analysed in [Brosgol, 1998].

Similarly to Ada, in C++11 class `thread` performs the given computation asynchronously. The computation should not end in an exception, otherwise the program is terminated.

There are also other concurrent object-oriented languages, which resolve concurrent exception problems by defining that exceptions must always be handled locally inside each thread of execution [Stoutamire and Omohundro, 1996].

Limiting exceptions to one thread is an easy way to solve the problem of concurrent exceptions, but it introduces its own problems. Both terminating the thread and ignoring an uncaught but important exception are unwanted and potentially fatal actions, so they should never occur. This forces the programmers to rely on older, more primitive exception signalling methods like special return values and status flags to propagate information on exceptional situations from a thread.

7.3.2 Exceptions in futures

Futures are an intuitive and widespread mechanism for asynchronous return value passing, and since exceptions can be regarded as an alternative to a return value, embedding exceptions to futures is a logical choice.

All Java inter-thread communication happens either through shared objects or synchronous method calls (like RMI, Remote Method Invocation). Java 5 also has classes `FutureTask` and `Future` to perform asynchronous computation. The mechanism is similar to the future mechanism presented in this thesis, except for future sources, future groups and exception reduction, which are not supported in Java 5.

Even though Java does not provide asynchronous method calls by default, there are several extensions to the language for this purpose. One of these is Java ARMI (Asynchronous Remote Method Invocation) [Raje *et al.*, 1997], which is built on normal synchronous Java RMI. ARMI uses futures to represent the results of asynchronous calls. For exception handling, ARMI provides two alternative methods.

The Delayed Delivery Mechanisms (DDM) embeds exceptions inside futures. The exceptions are thrown from the future when the return value of the asynchronous call is requested.

ProActive [Baduel *et al.*, 2006] is a GRID Java library for parallel, distributed, and concurrent programming. It provides active objects, asynchronous calls, and futures. Exceptions are handled synchronously, but [Caromel and Chazarain, 2005] describes a mechanism for concurrent exception handling. In their approach, futures receive concurrent exceptions and throw then when the value of a future is accessed. In addition to this, program execution waits at the end of a try block until all futures created inside that block have received their value. Possible exceptions are thrown at that point, making sure they can be handled in the try block surrounding the call that returned the future. In case of multiple exceptions, the approach throws the first one and ignores the rest. This is mentioned as problematic in the article, but the writers argue that also in a sequential program only the first exception would be thrown, because program execution would be transferred to the exception handler after throwing the exception.

The Argus language implements asynchronous calls using *call-streams* and uses a very future-like construct called a *promise* to handle return values from asynchronous calls [Liskov *et al.*, 1987, Liskov, 1988, Liskov and Shrira, 1988]. In Argus promises are strongly typed and represent the result of the asynchronous computation including possible exceptions. The type of the promise identifies the type of the return value and lists all possible exceptions which the promise may contain. Every asynchronous call returns a promise, which the caller can either poll periodically or start waiting for the call to complete ("*claim*" the promise). Waiting for the result of a promise either returns the normal return value of the call, or raises the exception the call has raised. If the same promise is claimed again, it re-returns the return value or re-raises the exception.

Version 4.4 of Qt [Qt Development Frameworks, 2008] introduced a class similar to KC++ `FutureGroup` called `QFutureSynchronizer`. Several futures can be registered to a synchroniser, and the synchroniser has a method for waiting for all futures to become ready. The destructor of the synchroniser also automatically synchronises with all futures, just like in KC++. Unlike in KC++, if a Qt future belongs to a `QFutureSynchronizer`, blocking and waiting for the value of the future does not synchronise with other futures belonging to the future synchroniser. Compared with KC++, Qt does not try to reduce exceptions coming from several futures.

It seems that developers of Qt have been somewhat aware of problems with concurrency and multiple exceptions. Normally, Qt futures throw arrived exceptions when methods requiring synchronisation are called. However, if exception handling is in progress, Qt futures do not throw exceptions. Qt uses C++ function `uncaught_exception()` to check this (Section 2.2.3 discusses its limitations). This behaviour of Qt futures is not documented. It is also dangerous, since Qt provides no other way to access exceptions in futures or to check whether a future contains exceptions. Current behaviour should be considered a bug, since it means that Qt futures cannot be safely used in code called from destructors. [Rintala, 2011]

In [Botinčan *et al.*, 2007], a mechanism similar to future groups called *future guards* is proposed. It allows multiple future synchronisation using an arbitrary boolean expression to express the required synchronisation condition. The paper does not discuss exception handling.

Using an arbitrary boolean expression to represent the synchronisation condition makes future guards a more expressive synchronisation mechanism than future groups presented in this thesis (which requires synchronisation with all futures in a group). However, exception reduction could become more complicated if exceptions are received only from some of the futures.

C++11 also uses futures for concurrent calls. Function `async` performs a given computation asynchronously. The return value of the computation is returned in a future, which also receives the possible exception. In addition t concurrency, `async` can be set to perform the computation in a lazy manner instead of concurrently. In that case, the futures perform the computation when its value is requested. If an exception occurs in lazy computation, it is thrown immediately. Future groups as a synchronisation mechanism have also been proposed for C++11, but they were not included in the language [Hinnant, 2006].

μC++ futures are similar to other futures described in this section. The only difference is that use of futures is explicit and exceptions end up in futures only if they are explicitly stored there. By default μC++ exceptions use asynchronous propagation, which is discussed in Section 7.3.4. μC++ also provides a mechanism for selectively waiting for a group of futures. This mechanism is similar to future guards but more flexible.

As mentioned earlier, embedding exceptions in futures is logical if exceptions are regarded as a special return value. Since exceptions directly alter the control flow of the program while return values do not, many systems using this approach

mention potential problems (as discussed in Section 5.2). Even though potential problems are mentioned, most systems do not try to solve the problems, but leave that to the programmer. The reduction mechanism described in this thesis is an attempt at providing tools for solving these problems.

If a program needs to react to multiple exceptions originating from futures without support from the language or system, it is important that the program can analyse exceptions stored in futures. This is easier if the interface of futures allows the program to query whether the future contains an exception, and access the exception without throwing it (this is logical if exceptions are regarded as alternate return values). Some future-based systems provide this interface, but not all. For example, in C++11 the only way to know if a future contains an exception is to ask for the content of the future, causing the exception to be thrown. This makes analysing and combining several exceptions difficult.

7.3.3 Exception callback functions

One solution to concurrent exception propagation is to allow a program to register callback functions or methods, which are automatically called when an exception of suitable type is received from another thread of execution.

Java ARMI provides such a mechanism (Callback Mechanism, CM in ARMI). It allows the programmer to attach special exception handlers to a future. Each exception handler is capable of handling a specific type of exception. If that exception occurs, ARMI automatically calls the appropriate exception handler.

JR [Keen *et al.*, 2001, Olsson and Keen, 2004] is another asynchronous extension to Java. JR implements asynchronous calls via *send* and *forward* statements. Exceptions are handled by requiring that each *send* and *forward* statement also specifies a *handler object*. Handler objects must implement the `Handler` interface and provide a method for each possible exception type, with the exception object as a parameter. When an exception occurs in an asynchronous call, an appropriate method in the handler object is called. The handler methods cannot throw any additional exceptions and they have to be able to completely handle the exceptional situation. The JR compiler statically checks that the handler object is able to handle all possible exceptions the call may throw. [Keen and Olsson, 2002, Chan *et al.*, 2005]

The callback mechanism is analysed in [Caromel and Chazarain, 2005]. It does give the programmer freedom to implement custom behaviour on exceptions, but

does not provide the ability to unwind the call stack on exceptions. Like try-catch blocks, it also separates exception handling code from its context.

In case of multiple concurrent exceptions, the callback mechanism does not give the programmer any control over how to combine or prioritise these exceptions. Typically, callbacks are called in some fixed order, making it difficult to respond to all of them in a sensible manner.

7.3.4 Asynchronous exception propagation

Asynchronous signalling has been present for a long time in system programming languages, where it is needed for signalling about CPU interrupts, operating system signals, etc. Since the source of such signals is concurrent from the program point of view, it is natural that similar mechanisms are also used in concurrent exception handling.

In many system programming languages like C and C++ asynchronous signalling is supported by letting the program register *signal handlers* for different asynchronous signals. This approach is similar to the callback approach discussed in Section 7.3.3.

Ada-95 defines an *asynchronous* accept statement. This is not an asynchronous call, but rather a way to execute a sequence of statements while waiting for an event to occur, like a rendezvous call, time expiration, or external signal. When the event happens, normal execution of statements is immediately aborted. This mechanism does not suffer from the exception problems of asynchronous calls, but introduces its own problems, for example because the abortion mentioned can occur in the middle of exception handling.

The approach used in μC++ differs from other future-based systems described in this chapter. As an alternative to embedding exceptions in futures, an exception resulting from an asynchronous call can trigger exception handling in the client at any time. Since there can clearly be parts of code where this is undesired, μC++ provides special syntax for selectively enabling and disabling the propagation exceptions based on their type. If a μC++ program is executing a part of code where a disabled exception is raised, handling of that exception is delayed until the exception becomes enabled again. Since the client may be blocked in synchronisation when an exception is raised, μC++ has special mechanisms for unblocking the client in these cases [Krischer and Buhr, 2008]. If multiple exceptions are raised

concurrently and end up in the same thread, they "are delivered serially" in μC++ [Buhr, 2010, Ch. 5].

Java also used asynchronous exception propagation when a thread was externally stopped by calling its `stop()` method. This caused an exception of type `ThreadDeath` to be thrown in the thread, interrupting its normal execution. Use of this mechanism is currently deprecated in Java, because of problems mentioned later in this section. The Java virtual machine can still throw asynchronous exceptions under fatal error conditions.

Asynchronous exception propagation suffers from the fact that it may trigger exception handling in the client regardless of where in the code the client is executing. This can be seen from the fact that many systems provide signal or exception masks to disable asynchronous propagation for a while, protecting a critical block of code from being interrupted. However, this means that programmers have to actively be aware of possible asynchronous exceptions and explicitly protect parts of code from them.

Asynchronous propagation also suffers from problems with multiple exceptions. If an asynchronous exception interrupts program execution, another asynchronous exception may interrupt the handling of the first exception, unless exception masking is used to disable further exceptions. Even in that case disabled exceptions are raised later when masking is removed. Delays caused by masking mean that multiple exceptions may be received in an undeterministic order, making it difficult to handle exceptions collectively. In summary, asynchronous exception propagation suffers from the same problems as other concurrent exception strategies, but has some additional problems of its own.

7.3.5 Other approaches

Besides the mechanisms discussed earlier in this chapter, there are of course many other ways to respond to concurrent exceptions.

In Ada, communication among tasks happens using the *rendezvous* mechanism, where one task calls a service on another task, which explicitly accepts the call. If an exception occurs during the rendezvous, the exception is propagated from the accepting task to the calling task (as well as propagated within the called task). However, this causes no problems in the calling task because normally rendezvous is a synchronous operation, so the calling task is waiting for the call to complete.

Forcing the exception propagation to be synchronous hides the problem of multiple exceptions, since the program must explicitly decide the order in which it synchronises with other threads. If a local exception is raised before synchronisation, it is handled first before any synchronisation is attempted. However, if the proper behaviour of the program requires knowledge of multiple exceptions originating from multiple sources, collecting this information is difficult if each exception requires explicit synchronisation and that synchronisation automatically raises the exception.

RMIX [Kurzyniec and Sunderam, 2004] is another Java RMI based communication framework providing asynchronous calls. In RMIX, if an asynchronous call by a client raises an exception in the server, the server refuses to accept further calls from the client until the client receives the exception through a synchronous call or until the server is manually released.

Although multiple exceptions are not explicitly mentioned, the explicit mechanism used in RMIX gives a client freedom to choose which exceptions it wants to receive and in which order. This again theoretically gives the program freedom to receive and react to multiple exceptions, although such a program can be very difficult to write.

In the programming language Arche [Benveniste and Issarny, 1992], asynchrony and synchronisation have been implemented quite differently. Every object in the language has its own thread of control and every method call is synchronous. However, concurrent objects can communicate and co-operate asynchronously using *multimethods* (invocation of a method in a group of objects). Exception handling problems are solved by attaching to each multimethod a *coordinator*, which controls the overall action. The coordinator may have a *resolution function* which receives exceptions from all participants and computes a *concerted exception* representing the resulting "total" exception [Issarny, 2001]. The initial idea of the resolution trees has been developed in [Campbell and Randell, 1986].

The multimethod approach clearly acknowledges the fact that concurrent exception handling is more complex than sequential exception handling. It allows exception handling to collect exception information from multiple threads (like in exception reduction discussed in this thesis), and the results of exception handling affect all participating threads. On the other hand, multimethods are not part of most object-oriented programming languages, so the Arche approach cannot easily be used in other languages.

7.3.6 Summary

Table 7.1 on the following page shows a summary of concurrency and exception handling features of systems discussed in this section. A check mark means that a feature is found, a cross means that it is not. A dash means that a feature is irrelevant (for example, if exceptions are not supported, features based on exceptions are irrelevant). Some systems support several concurrency mechanisms, so seemingly mutually exclusive features may be marked for a system if one of its concurrency mechanisms support one feature and another mechanism support the other.

The table shows that of the systems mentioned in this chapter, there is no clear pattern on how concurrent exceptions are handled, even though futures are commonly used to transfer them. This applies to both systems which support threads (a shared address space) and distributed or process based systems with no shared memory.

	C++ family					Java family					Others		
Feature	C++11	Qt	μC++	C++//	KC++	Java	Java ARMI	JR	Pro-Active	RMIX	Ada-95	Argus	Arche
Shared address space	✓	✓	✓	×	×	✓	×	×	×	×	✓	✓	×
Separate address spaces	×	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Futures	✓	✓	✓	✓	✓	✓	✓	×	✓	✓	×	✓	×
Multiple future synchronisation	×	✓	✓	×	✓	×	×	–	×	×	×	×	–
Exceptions from futures	✓	✓	✓	–	✓	✓	✓	–	✓	✓	×	✓	–
Exception callbacks	×	×	×	–	×	×	✓	✓	×	✓	×	×	×
Asynchronous exception propagation	×	×	✓	–	×	✓	×	×	×	×	✓	×	×
Multiple exceptions terminate	✓	×	×	–	×	×	×	×	×	×	✓	×	×
Multiple exceptions ignored	×	✓	×	–	×	✓	×	×	✓	✓	✓	✓	×
Multimethods	×	×	×	×	×	×	×	×	×	×	×	×	✓
Exception reduction	×	×	×	–	✓	×	×	×	×	×	×	×	✓

✓ = Is supported
× = Is not supported
– = Does not apply (e.g., no futures or exception handling)

TABLE 7.1: *Summary of concurrency and exception features*

Chapter 8

Conclusion

This thesis has provided mechanisms for concurrent exception handling in the C++ language. This includes automatic serialisation of exceptions for RPC as well as handling multiple concurrent exceptions in concurrent programs. This chapter gives a summary of the contributions of this thesis and discusses limitations of the mechanisms. It also analyses the applicability of those mechanisms for languages other than C++ and finally presents some ideas for future work.

8.1 Contributions revisited

This section revisits the main contributions presented in this thesis.

- **Automated serialisation of RPC exceptions**

  In this thesis it is shown that exception propagation in RPC and similar calls with no shared memory can be implemented as a template-based library, including serialisation and dynamic creation of exception objects. The mechanism is also suitable for throwing and catching exceptions using an exception class hierarchy.

  The solution requires minimal additional code from application programmers and allows the use of existing exception hierarchies. The mechanism has been implemented and tested in the KC++ system developed by the author.

  The presented solution is light-weight and implemented completely using C++ and its template metaprogramming mechanisms. Necessary object factor-

ies and virtual functions are generated automatically, which means no preprocessors, code generators or separate IDL specifications are needed.

The mechanism can be used with third-party exception classes, which cannot be modified to include methods for serialisation, dynamic creation, and dynamic throwing. A mechanism for automatic mapping of third-party exceptions to RPC-enabled counterparts is presented.

- **Handling and reduction of multiple concurrent exceptions**

  This thesis shows how support for multiple exceptions in asynchronous concurrent calls can be added to C++. An analysis of problems related to concurrency and exceptions is presented, using the C++ language as an example. A solution based on exception reduction, future groups and compound exceptions is presented.

  Exception reduction is further developed by presenting a template metaprogramming based framework for exception reduction, where a reduction strategy can be combined from ready-made components. Exception reduction based on inheritance hierarchies is also discussed and an implementation is presented. Limitations of the mechanisms and the underlying C++ language are discussed.

  The mechanisms presented in the thesis have also been implemented in the KC++ system.

- **Performance analysis and a case study**

  A performance test for the RPC exception passing mechanism was executed and the results are discussed in this thesis. Performance tests show that the efficiency of the RPC exception passing mechanism in this thesis is good in situations where exceptions themselves are an acceptable mechanism. An RPC exception propagation takes about 40 times longer than normal RPC return. This is considered acceptable, especially taking into account that without RPC, normal exception handling in C++ is 3–4 degrees of magnitude slower than normal return from a function.

  Usability of the exception reduction mechanism was evaluated with a case study. A concurrent version of the Observer pattern was implemented in KC++ and special emphasis was put on exception handling. This thesis contains a discussion about issues found during the implementation.

The Observer pattern was then used to implement a simple concurrent image processing application. Exception handling and reduction in this application are discussed in the thesis, including problems found during the case study.

A simple performance test was also performed on the case study to get an idea of the performance of exception reduction. For this purpose, several versions of the image processing application were compiled and these versions were compared with each other. Performance results are discussed in this thesis, suggesting that exception reduction mechanisms presented in this thesis are acceptable from the performance standpoint.

8.2 Future research and work

The work in this thesis is not complete. There are several obvious directions for future research, some of which are independent of each other.

The largest and broadest research direction is concurrent exception resolution in general. Existing software projects using concurrency could be analysed to see what strategies have been chosen to solve problems with multiple exceptions (or what problems are found because multiple exceptions have not been taken into account). Such research could provide information for writing a more complete library of ready-made components for exception reduction. It could also reveal problems where exception reduction is not an adequate solution for solving multiple exception problems.

Research in this thesis has shown that combining exception handling and concurrency is not trivial. Traditional exception handling is based on providing an alternative control flow when an exception occurs. This becomes problematic when the program consists of several concurrent flows of control. These problems should be futher analysed to see whether changes are needed in programming languages' exception handling semantics in a concurrent environment. Similar problems occur with asynchrony caused by lazy evaluation. One research direction is to analyse existing lazy programming languages and their mechanisms for exception and error handling, and how those could be adapted for concurrent exception handling in eager languages (like C++).

Future groups group together several futures and perform synchronisation and exception reduction among them. In the current version future groups wait for all futures to become ready to make sure all possible exception participate in reduction.

Other options should be considered as well in future research. For example, it would be useful to be able to wait for the *first* exception arising from a future group, or to stop waiting after a timeout. In a more general case it would be useful to analyse currently received exceptions and decide whether to wait for the rest or proceed with exception reduction.

One direction for future work comes from development in the C++ language itself. KC++ and mechanisms presented in this thesis have been written before the new C++11 standard. The new language standard provides additional support for template metaprogramming as well as tools for concurrency, and incorporating those into KC++ and using them to improve concurrent exception handling are obvious areas for future work.

Of most interest is concurrency support in C++11. Concurrency is based on threads executing in the same address space, so RPC exception passing mechanisms like the one in this thesis are still needed. Superficially, it seems that C++11 does not provide any new tools for serialisation of exceptions, but this should be verified with further research.

Since futures will now be part of the C++11 standard library, one future direction is to change exception reduction mechanisms to work with C++11 futures, if possible. Ideally, this could work without modifying or extending C++11 futures, but that is not known without additional research.

C++11 provides some new tools which could make the implementation of mechanisms in this thesis easier. One such feature is *variadic templates*, which are templates with variable and unlimited number of template arguments. There are several places in this thesis where the programmer should be able to provide an arbitrary list of classes. Currently, lots of template metacode are needed to allow this, and even then the maximum number is limited. Rewriting this code to use variadic templates is an obvious place for future work.

Finally, C++11 extends the C++ language with lambda functions, making it easy to write small utility functions, which can also have access to their environment. This would be useful for writing reduction functions. Lambda functions combined with variadic templates could also be used to improve reduction combinators. However, there are limitations in combining lambdas and templates, so further research is necessary.

8.3 Concluding remarks

This thesis has shown how concurrent exception support in C++ can be built as a library, without changing the underlying language. Problems in handling multiple concurrent exceptions are discussed and analysed, and a solution using exception reduction is presented. Viability of the solutions are verified with performance analysis and a case study.

During the research included in this thesis it has become clear to the author that concurrent exception handling is not trivial and that exception handling support of most current programming languages is not adequate to solve all emerging problems. This observation is made more interesting by the fact that concurrent programming is rapidly becoming more and more mainstream.

It has also become clear that designing a concurrent exception handling strategies for applications is even more difficult than designing traditional sequential exception handling. Existing practises, design patterns and guidelines for exception handling do not necessarily work in a concurrent environment.

Bibliography

[Abrahams and Gurtovoy, 2004] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[Abrahams *et al.*, 2006] David Abrahams, Steve Cleary, Beman Dawes, Aleksey Gurtovoy, Howard Hinnant, Jesse Jones, Mat Marcus, Itay Maman, John Maddock, Alexander Nasonov, Thorsten Ottosen, Robert Ramey, Jeremy Siek, and Adobe Systems Inc. Boost.Typetraits. `http://www.boost.org/doc/libs/1_47_0/libs/type_traits/doc/html/index.html`, 2006. Checked Sep 2011.

[Ada, 1995] Intermetrics, Inc. *Ada 95 Reference Manual*, December 1995.

[Adler *et al.*, 2002] Darin Adler, Greg Colvin, and Beman Dawes. The Boost smart pointers. `http://www.boost.org/doc/libs/1_47_0/libs/smart_ptr/smart_ptr.htm`, 2002. Checked Sep 2011.

[Adobe Systems, Inc., 2008] Adobe Systems, Inc. Adobe Photoshop Lightroom. `http://www.adobe.com/products/photoshoplightroom/`, May 2008. Checked May 2008.

[Alexandrescu, 2001] Andrei Alexandrescu. *Modern C++ Design*. C++ In-Depth Series. Addison-Wesley, 2001.

[Andrews, 1991] Gregory R. Andrews. *Concurrent Programming — Principles and Practice*. Addison-Wesley, Reading, Massachusetts, 1991.

[Arnold and Gosling, 1998] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1998.

[Baduel *et al.*, 2006] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[Bartosik, 2004] Mark Bartosik. Templates & marshaling C++ function calls. *C/C++ Users Journal*, 22(7):28–33, July 2004.

[Baude *et al.*, 1996] Françoise Baude, Fabrice Belloncle, Denis Caromel, Nathalie Furmento, Yves Roudier, Philippe Mussi, and Günther Siegel. Parallel object-oriented programming for parallel simulations. *Information Sciences*, 93(1–2):35–64, August 1996.

[Becker, 2007] Pete Becker. *The C++ Standard Library Extensions—A Tutorial and Reference*. Addison-Wesley, 2007.

[Benveniste and Issarny, 1992] M. Benveniste and V. Issarny. Concurrent programming notations in the object-oriented language Arche. Research Report 1822, INRIA, Rennes, France, 1992.

[Botinčan *et al.*, 2007] Matko Botinčan, Davor Runje, and Albert Vučinović. Futures and the lazy task creation for .net. In *Software, Telecommunications and Computer Networks, 2007. SoftCOM 2007. 15th International Conference on*, pages 1–5, September 2007.

[Brosgol, 1998] Benjamin M. Brosgol. A comparison of the concurrency features of Ada 95 and Java. In *Proceedings of the 1998 annual ACM SIGAda international conference on Ada*, pages 175–192, Washington, D.C., United States, 1998. ACM Press.

[Bryant and O'Hallaron, 2003] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2003.

[Budd, 2002] Timothy Budd. *An Introduction to Object-oriented Programming, 3rd edition*. Addison-Wesley, Reading, Massachusetts, 2002.

[Buhr and Mok, 2000] Peter A. Buhr and W. Y. Russel Mok. Advanced exception handling mechanisms. *IEEE Transactions on Software Engineering*, 26(9):820–836, September 2000.

[Buhr *et al.*, 1992] P. A. Buhr, G. Ditchfield, R. Stroobosscher, and B. Younger. μC++: Concurrency in the object-oriented language C++. *Software Practice and Experience*, 22(2), February 1992.

[Buhr, 2010] Peter A. Buhr. *μC++ Annotated Reference Manual, version 5.7.0*. University of Waterloo, 2010.

[Campbell and Randell, 1986] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, 12(8):811–826, 1986.

[Caromel and Chazarain, 2005] Denis Caromel and Guillaume Chazarain. Robust exception handling in an asynchronous environment. In Romanovsky et al. [2005], pages 2–14. Technical Report No 05-050.

[Caromel *et al.*, 1996] Denis Caromel, Fabrice Belloncle, and Yves Roudier. C++//. In Wilson and Lu [1996], pages 257–296.

[Chan *et al.*, 2005] Hiu Ning (Angela) Chan, Esteban Pauli, Billy Yan-Kit Man, Aaron W. Keen, and Ronald A. Olsson. An exception handling mechanism for the concurrent invocation statement. In *Euro-Par 2005: Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 699–709. Springer-Verlag, 2005.

[Chatterjee, 1989] A. Chatterjee. Futures: a mechanism for concurrency among objects. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 562–567. ACM/IEEE, ACM Press, 1989.

[CodeSourcery, 2001] CodeSourcery. Itanium C++ ABI. `http://www.codesourcery.com/cxx-abi/`, March 2001. Checked Sep 2011.

[Collins, 1960] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3:655–657, December 1960.

[Coplien, 1992] James O. Coplien. *Advanced C++*. Addison-Wesley, Reading, Massachusetts, 1992.

[Coplien, 1995] James O. Coplien. Curiously recurring template patterns. *C++ Report*, February 1995.

[Czarnecki and Eisenecker, 2000] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming — Methods, Tools, and Applications*. Addison-Wesley, Reading, Massachusetts, 2000.

[de Guzman and Kaiser, 2011] Joel de Guzman and Hartmut Kaiser. Spirit 2.4.2. http://www.boost.org/doc/libs/1_46_1/libs/spirit/, 2011. Checked May 2011.

[Dijkstra, 2002] Edsger W. Dijkstra. Cooperating sequential processes. In Per Brinch Hansen, editor, *The origin of concurrent programming*, pages 65–138. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[Edison Design Group, 2011] Edison Design Group. The C++ front end. http://www.edg.com/index.php?location=c_frontend, 2011. Checked Sep 2011.

[FLTK, 2008] Fast Light Toolkit. http://www.fltk.org/, May 2008. Checked May 2008.

[Free Software Foundation, Inc., 2004] Free Software Foundation, Inc. GNU Common C++. http://www.gnu.org/software/commoncpp/, 2004. Checked Jan 2004.

[Friesenhahn, 2007] Bob Friesenhahn. Magick++ – C++ API for ImageMagick. http://www.imagemagick.org/Magick++/, September 2007. Checked May 2008.

[Gamma *et al.*, 1996] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, Reading, 1996.

[GNU, 2011] GNU. The GNU compiler collection (GCC). http://gcc.gnu.org/, 2011. Checked Sep 2011.

[Goetz, 2005] Brian Goetz. Java theory and practice: Be a good (event) listener. http://www.ibm.com/developerworks/java/library/j-jtp07265/index.html, July 2005. Checked Sep 2011.

[Goodenough, 1975] John B. Goodenough. Structured exception handling. In *Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 204–224, New York, NY, USA, January 1975. ACM Press.

[Gosling *et al.*, 2005] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, Reading, Massachusetts, 2005.

[Gruntz, 2002] Dominik Gruntz. Java design: On the observer pattern. *Java Report*, February 2002. Accepted for publication, but Java Report ended in fall 2001. Pdf version available `http://www.gruntz.ch/papers/Observer.pdf`. Checked Sep 2011.

[Gurtovoy and Abrahams, 2004] Aleksey Gurtovoy and David Abrahams. The Boost MPL library. `http://www.boost.org/libs/mpl/doc/index.html`, 2004. Checked Sep 2011.

[Halstead, 1985] Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Henney, 2002] Kevlin Henney. From mechanism to method: The safe stacking of cats. *C/C++ Users Journal Experts Forum*, February 2002. `http://accu.org/index.php/journals/235`. Checked Sep 2011.

[Hinnant, 2006] Howard E. Hinnant. Multithreading API for C++0x—a layered approach. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2094.html`, September 2006. C++ Standards Committee Document No. N2094=06-0164. Checked Sep 2011.

[Hoare, 1974] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[ImageMagick Studio LLC, 2008] ImageMagick Studio LLC. ImageMagick. `http://www.imagemagick.org/`, May 2008. Checked May 2008.

[ISO/IEC JTC1/SC22, 2006] ISO/IEC JTC1/SC22. Technical report on C++ library extensions. International standard under publication ISO/IEC TR 19768, ISO/IEC, June 2006.

[ISO/IEC WG21, 2006] ISO/IEC WG21. Technical report on C++ performance. Technical Report TR 18015:2006, ISO/IEC, 2006.

http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf. Checked Sep 2011.

[ISO/IEC, 2003] ISO/IEC. *ISO/IEC 14882:2003 – Programming Languages – C++, Second Edition*. ISO/IEC, October 2003.

[ISO/IEC, 2012] ISO/IEC. *ISO/IEC 14882:2011 – Programming Languages – C++*. ISO/IEC, February 2012.

[Issarny, 2001] Valérie Issarny. *Concurrent Exception Handling*, chapter 7, pages 111–127. Volume 2022 of Romanovsky et al. [2001], 2001.

[Järvi, 2000] Jaakko Järvi. *New Techniques in Generic Programming — C++ is More Intentional than Intended*. PhD thesis, University of Turku, Turku, Finland, May 2000. TUCS Dissertation #26.

[Jordan, 1978] Harry F. Jordan. A special purpose architecture for finite element analysis. In *Proceedings of the 1978 International Conference on Parallel Processing*, pages 263–266, 1978.

[Keen and Olsson, 2002] Aaron W. Keen and Ronald A. Olsson. Exception handling during asynchronous method invocation. In *Euro-Par 2002: Parallel Processing*, volume 2400 of *Lecture Notes in Computer Science*, pages 656–660. Springer-Verlag, 2002.

[Keen *et al.*, 2001] A. W. Keen, Tingjian Ge, J. T. Maris, and R. A. Olsson. JR: flexible distributed programming in an extended Java. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems*, pages 575–584. IEEE, April 2001.

[Krasner and Pope, 1988] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1:26–49, August–September 1988.

[Krischer and Buhr, 2008] Roy Krischer and Peter A. Buhr. Asynchronous exception propagation in blocked tasks. In *Proceedings of the 4th International Workshop on Exception Handling, WEH 2008*, pages 8–15. ACM, November 2008.

[Kurzyniec and Sunderam, 2004] D. Kurzyniec and V. Sunderam. Semantic aspects of asynchronous RMI: the RMIX approach. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 157, April 2004.

[Landry, 2003] Walter Landry. Implementing a high performance tensor library. *Scientific Programming*, 11:273–290, 2003.

[Lavender and Schmidt, 1995] R. Greg Lavender and Douglas C. Schmidt. Active object: An object behavioral pattern for concurrent programming. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Program Design 2*, chapter 30. Addison-Wesley, Reading, Massachusetts, 1995.

[Lee, 2006] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006. DOI: 10.1109/MC.2006.180.

[Liskov and Shrira, 1988] Barbara Liskov and Luiba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 260–267, June 1988.

[Liskov *et al.*, 1987] Barbara Liskov, Mark Day, Maurice Herlihy, Paul Johnson, Robert Scheifler, and William Weihl. Argus reference manual. Technical Report MIT-LCS-TR-400, Massachusetts Institute of Technology, 1987.

[Liskov, 1988] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[Magee and Kramer, 1999] Jeff Magee and Jeff Kramer. *Concurrency—State Models & Java Programs*. Wiley, 1999.

[McBeth, 1963] J. Harold McBeth. Letters to the editor: on the reference counter method. *Communications of the ACM*, 6:575, September 1963.

[Meyers, 1996] Scott Meyers. *More Effective C++*. Addison-Wesley, Reading, Massachusetts, 1996.

[Meyers, 1998] Scott Meyers. *Effective C++ 2nd edition*. Addison-Wesley, Reading, Massachusetts, 1998.

[Milner *et al.*, 1997] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[Nikon Corporation, 2008] Nikon Corporation. Capture NX. `http://www.capturenx.com/`, May 2008. Checked May 2008.

[Olsson and Keen, 2004] Ronald Olsson and Aaron Keen. *The JR Programming Language Concurrent Programming in an Extended Java*. Springer Science + Business Media, Inc., 2004. DOI: 10.1007/b116040.

[OMG, 1998] Object Management Group, Inc., 492 Old Connecticut Path, Framingham, MA 01701. *The Common Object Request Broker: Architecture and Specification*, 1998. version 2.2.

[OMG, 2003] Object Management Group. *C++ Language Mapping Specification*, June 2003. Version 1.1.

[OMG, 2004] Object Management Group. *Common Object Request Broker Architecture: Core Specification*, March 2004. Version 3.0.3.

[Ploski and Hasselbring, 2005] Jan Ploski and Wilhelm Hasselbring. The callback problem in exception handling. In Romanovsky et al. [2005], pages 39–50. Technical Report No 05-050.

[Qt Development Frameworks, 2008] Qt Development Frameworks. Qt reference documentation. `http://doc.qt.nokia.com/4.4/index.html`, May 2008. Checked Mar 2011.

[Qt Development Frameworks, 2011] Qt Development Frameworks. Qt—cross-platform application and UI framework. `http://qt.nokia.com/`, March 2011. Checked Mar 2011.

[Raje *et al.*, 1997] Rajeev R. Raje, Joseph Williams, and Michael Boyles. An asynchronous remote method invocation (ARMI) mechanism for Java. *Concurrency: Practice and Experience*, 9(11):1207–1211, November 1997.

[Ramey, 2004] Robert Ramey. The Boost serialization library. `http://www.boost.org/libs/serialization/doc/index.html`, November 2004. Checked Sep 2011.

[Reenskaug, 1979] Trygve Reenskaug. THING-MODEL-VIEW-EDITOR, an example from a planningsystem. Technical report, Xerox PARC, May 1979.

[Rintala, 2000] Matti Rintala. KC++ — a concurrent C++ programming system. Licentiate thesis, Tampere University of Technology, Tampere, Finland, 2000.

[Rintala, 2006] Matti Rintala. Handling multiple concurrent exceptions in C++ using futures. In C. Dony, J. Lindskov Knudsen, A. Romanovsky, and A. Tripathi, editors, *Advanced Topics in Exception Handling Techniques*, 4119 of Lecture Notes in Computer Science, pages 62–80. Springer-Verlag, 2006. DOI: 10.1007/11818502_4.

[Rintala, 2007] Matti Rintala. Exceptions in remote procedure calls using C++ template metaprogramming. *Software – Practice and Experience*, 37:231–246, 2007. DOI: 10.1002/spe.754.

[Rintala, 2011] Matti Rintala. QTBUG-18149 QFuture does not throw exceptions if called from destructors during stack unwinding. http://bugreports.qt.nokia.com/browse/QTBUG-18149, March 2011. Checked Sep 2011.

[Romanovsky and Kienzle, 2001] Alexander Romanovsky and Jörg Kienzle. *Action-Oriented Exception Handling in Cooperative and Competitive Object-Oriented Systems*, pages 147–164. Volume 2022 of Romanovsky et al. [2001], 2001.

[Romanovsky *et al.*, 2001] A. Romanovsky, C. Dony, J. Lindskov Knudsen, and A. Tripathi, editors. *Advances in Exception Handling Techniques*, volume 2022 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[Romanovsky *et al.*, 2005] A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors. *Developing Systems that Handle Exceptions, Proceedings of ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems*. Department of Computer Science, LIRMM, Montpellier-II University, 2005. Technical Report No 05-050.

[Romanovsky, 2000] Alexander Romanovsky. Extending conventional languages by distributed/concurrent exception resolution. *Journal of Systems Architecture*, 46(1):79–95, 2000.

[Ryder and Soffa, 2003] Barbara G. Ryder and Mary Lou Soffa. Influences on the design of exception handling. *ACM SIGPLAN Notices*, 38(6):16–22, June 2003.

[s11n, 2005] Homepage of s11n.net. http://s11n.net/, 2005. Checked Sep 2011.

[Stevens, 1992] W. Richard Stevens. *Advanced Programming in the UNIX(R) Environment*. Addison-Wesley, 1992.

[Stoutamire and Omohundro, 1996] David Stoutamire and Stephen Omohundro. Sather 1.1. `http://www.icsi.berkeley.edu/~sather/`, August 1996. Checked Sep 2011.

[Stroustrup, 1993] Bjarne Stroustrup. A history of C++: 1979–1991. *SIGPLAN Notices*, 28:271–297, March 1993.

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

[Stroustrup, 2000] Bjarne Stroustrup. *The C++ Programming Language (Special Edition)*. Addison-Wesley, Reading, Massachusetts, 2000.

[Stroustrup, 2007] Bjarne Stroustrup. Evolving a language in and for the real world: C++ 1991–2006. In Barbara Ryder and Brent Hailpern, editors, *Proceedings of the The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, pages 4/1–4/59. ACM, June 2007.

[Sun Microsystems, Inc., 2006] Sun Microsystems, Inc. Java™ 2 platform standard edition 6.0 API specification. `http://download.oracle.com/javase/6/docs/api/`, December 2006. Checked Sep 2011.

[Sutter and Alexandrescu, 2005] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards*. Addison-Wesley, Reading, Massachusetts, 2005.

[Sutter and Hyslop, 2005] Herb Sutter and Jim Hyslop. Polymorphic exceptions. *C/C++ Users Journal*, 23(4):58–60, April 2005.

[Sutter, 2000] Herb Sutter. *Exceptional C++*. C++ In-Depth Series. Addison-Wesley, Reading, Massachusetts, 2000.

[Sutter, 2001] Herb Sutter. *More Exceptional C++*. C++ In-Depth Series. Addison-Wesley, 2001.

[Sutter, 2008] Herb Sutter. Interrupt politely. *Dr Dobb's Journal*, May 2008. `http://ddj.com/architect/207100682`. Checked Sep 2011.

[Szallies, 1997] Constantin Szallies. On using the observer design pattern. http://www.wohnklo.de/patterns/observer.html, August 1997. Checked Sep 2011.

[Szyperski *et al.*, 2002] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software—Beyond Object-Oriented Programming, 2nd edition*. Addison Wesley / ACM Press, 2002.

[Vandevoorde and Josuttis, 2003] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates — The Complete Guide*. Addison-Wesley, 2003.

[Veldhuizen, 1998] Todd Veldhuizen. Arrays in blitz++. In Denis Caromel, Rodney Oldehoeft, and Marydell Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, pages 501–501. Springer Berlin / Heidelberg, 1998. 10.1007/3-540-49372-7_24.

[Veldhuizen, 2000] Todd L. Veldhuizen. Techniques for scientific C++. Technical Report 542, Indiana University, Computer Science, August 2000.

[Veldhuizen, 2003] Todd L. Veldhuizen. C++ templates are turing complete. Technical report, Indiana University, 2003.

[Wikipedia, 2011] Curiously recurring template pattern. http://en.wikipedia.org/wiki/Curiously_recurring_template_pattern, March 2011. Checked Mar 2011.

[Williams, 2008] Anthony Williams. The Boost thread library. http://www.boost.org/doc/libs/release/doc/html/thread.html, 2008. Checked Sep 2011.

[Wilson and Lu, 1996] G. Wilson and P. Lu, editors. *Parallel Programming Using C++*. MIT Press, Cambridge (MA), USA, 1996.

[Wilson, 1992] Paul Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 1–42. Springer Berlin / Heidelberg, 1992. 10.1007/BFb0017182.

[Winder *et al.*, 1996] Russel Winder, Graham Roberts, Alistair McEwan, Jonathan Poole, and Peter Dzwig. UC++. In Wilson and Lu [1996], pages 629–670.

[Xu *et al.*, 2000] Jie Xu, A.Romanovsky, and B. Randell. Concurrent exception handling and resolution in distributed object systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1019–1032, October 2000.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland

ISBN 978-952-15-2915-3
ISSN 1459-2045