

Active Objects and Futures: A Concurrency Abstraction Implemented for C[#] and .NET

Bachelorarbeit

vorgelegt von
Tobias Gurock

Betreuer: Dr. Michael Thies
Gutachter: Prof. Dr. Uwe Kastens

Universität Paderborn
Fakultät für Elektrotechnik, Informatik und Mathematik
Arbeitsgruppe Programmiersprachen und Übersetzer

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig und ohne unerlaubte fremde Hilfe sowie ohne Benutzung anderer als den angegebenen Quellen angefertigt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Paderborn, im Dezember 2007

Contents

1	Introduction	1
2	Concepts	3
2.1	Active Objects	3
2.1.1	Components	5
2.2	Futures	7
2.2.1	Errors	8
2.2.2	States	9
2.2.3	Grouping	9
2.3	Active Blocks	10
2.4	Related Concepts	11
2.4.1	Remote Objects	11
2.4.2	Active Monitors	12
3	Design of Implementation	13
3.1	Runtime	13
3.1.1	Dispatcher	13
3.1.2	Future	15
3.1.3	Operation	16
3.1.4	Tasks	17
3.2	Compiler	17
3.3	Examples	18
3.3.1	Active Objects	18
3.3.2	Future Groups	20
3.3.3	Active Blocks	21
3.4	Requirements	22
3.5	Comparison	23
3.5.1	Java	23
3.5.2	C++	25
4	Proposal for C[#] Extension	27
4.1	Concepts	27
4.1.1	Active Objects	27

4.1.2	Futures	30
4.1.3	Active Blocks	34
4.2	Benefits	36
5	Evaluation	37
5.1	Abstraction	37
5.2	Uses	38
5.2.1	Algorithms	39
5.2.2	Data Structures	40
5.2.3	Asynchronous I/O	42
5.2.4	User Feedback	42
5.3	Performance	43
5.4	Future Work	44
5.4.1	Active Object Pools	45
5.4.2	Scalability	45
5.4.3	Efficiency	46
5.4.4	Dynamic Priorities	47
5.5	Limitations	47
6	Conclusion	49

Chapter 1

Introduction

In the recent years, concurrency features like Hyper-Threading or multi-core functionality have found their ways into standard desktop computers. The hardware trend is thus changing from getting *faster* processors to getting *more* processors [23]. This marks a significant change in how software will be developed in the future. Whereas hardware performance improvements have been taken for granted in the past with hardware – especially processors – getting faster and faster every year, developers are now forced to take care that their applications are designed to adapt well to future hardware. Assuming that a software application will automatically run faster with a new processor model or architecture is thus no longer valid. Developers need to make use of this new hardware model and its features explicitly by introducing concurrency and especially multithreading into their applications.

Although being touted as the *next big thing* for years now, concurrent programming is still very difficult and error-prone even in modern programming languages and environments. Concurrent and in particular multithreaded programming still requires deep knowledge of low-level techniques like threading or locks and experience with deadlocks, race conditions and other subtle and hard to find concurrency problems. High-level concurrency abstractions and patterns can simplify concurrent programming [25] but are only starting to become available in modern mainstream programming languages.

One of these high-level abstractions is the so called *Active Object* [12] pattern. The Active Object pattern works, as the name implies, on the object level and not on an object hierarchy like most other design patterns [5]. With an active object, method invocation is decoupled from the actual method execution, i.e. invoked methods of these objects are executed asynchronously and do not block the caller. There are several variants of this pattern known, but all have in common that the concurrency functionality is achieved by running methods in a thread or process context different from that of the caller. Possible results of active methods are encapsulated in so called *future* [17] objects which can be seen as placeholders or contracts for the real results. A future is basically a join point or rendezvous for the caller and the active object. Once a result of a method is computed, the active object stores it in the

related future object and the caller can access it there. If the caller tries to access the result before the method is completed, the caller automatically blocks until it is available.

This thesis presents and describes the Active Object concurrency pattern in detail and provides an example implementation for .NET (hereinafter referred to as *active object runtime library*) written in C[#] [18, 19, 15]. I start in chapter 2 with introducing the theoretical aspects of the Active Object pattern. This chapter lays out the foundation for the rest of this paper by presenting the fundamental properties and characteristics of active objects and futures as well as the underlying models. Also covered in this chapter is the related concept of *active blocks* [24] which basically represent a block of code that can be executed in parallel as a whole. I conclude this chapter with giving a summary of some technologies and concepts which are related to the Active Object model.

The subsequent chapter 3 is dedicated to the practical part of the thesis. I begin with summarizing the developed runtime library for active objects and its implementation and then introduce the *active object compiler*, a code generator tool that reduces the repetitive tasks involved in writing active objects and simplifies using the runtime library. It then follows a section which demonstrates the capabilities of the runtime library and its underlying implementation in practice by means of several examples. Also included in this chapter is a discussion of the required minimum features to provide a comparable concurrency abstraction in other programming languages and environments and a comparison of C[#] and .NET with other object-oriented languages and environments such as C++ and Java.

Chapter 4 goes on by presenting a proposal for an optional extension for the C[#] compiler itself which mimics the functionality of my active object compiler and makes active objects, futures and active blocks first class language constructs. After introducing the basic concept of this language extension and providing a complete reference about the required C[#] language changes, I conclude this chapter by pointing out the advantages and disadvantages of integrating the Active Object model directly into the programming language over the standard approach of using a compiler-independent add-on library.

Chapter 5 then evaluates the Active Object model. It presents it as a general high-level abstraction for traditional threading models and APIs and gives several typical use-case scenarios for active objects, futures and active blocks. I then go on by discussing the performance implications of the Active Object model. After that, it follows a section on possible advanced future improvements which are neither implemented in the example runtime library nor mentioned in my C[#] language extension proposal. I then conclude this chapter with the limitations and shortcomings of the Active Object model.

The final chapter 6 then concludes this thesis by summarizing my work and highlighting its main parts.

Chapter 2

Concepts

There are two different and independent approaches in the Active Object model for adding concurrency to applications, namely the *object-level* and *operation-level* concurrency models.

Concurrency on the object-level involves modeling classes as active classes with the implications that their operations are processed asynchronously and, which is equally important in most cases, inherently thread-safe with respect to each other by processing at most one operation at any given time. Adding concurrency on the operation-level on the other hand means that individual operations which are especially suited for parallel processing are explicitly designed to be carried out in parallel. Operation-level concurrency in the Active Object pattern is achieved by means of active blocks.

In this chapter, I introduce the fundamental concepts and properties of the Active Object pattern and both concurrency models. I start with the basic behavior and underlying structure of active objects and proceed with explaining the functionality and capabilities of the concept of futures as well as of active blocks. I then conclude this chapter with giving a short summary of some technologies and concepts which are related to the Active Object model.

The concepts and terminology presented, described and used throughout this chapter and the whole thesis are influenced by and based on [12, 24]. Schmidt and Lavelander [12] propose an Active Object concept and implementation for C++ which is very similar to my interpretation. The terminology of this thesis is primarily based on this paper. Sutter goes further with his Concur Project [24] and introduces active blocks as well as groups of futures. The Active Object model as presented in this chapter can roughly be seen as a combination of both views.

2.1 Active Objects

Active objects are very similar to traditional objects (also called *passive objects* in this context for a better differentiation). They have private fields and provide methods to operate on this data. The only important difference lies in the fact that each

active object runs in its own thread of control and that invoked methods do not block the caller but are executed asynchronously. Methods of active objects as presented in this thesis are always executed in a single thread and run sequentially with respect to each other, i.e. it is not possible that two method calls of one particular active object run at the same time. This simple condition automatically guarantees that the implementation of an active object does not require any additional mechanisms to ensure thread-safety for this particular object.

To understand how active objects work, it is important to differentiate between their public interface and their internal operation. The public interface of an active object is often called *proxy* and the internals are represented by a so called *servant* or *implementation*. The proxy is responsible for accepting method calls or *requests* from *clients*, other objects (passive or active) which utilize an active object. Public method requests to an active object and possible arguments are *marshaled* or converted into *messages* by the proxy and added to a *message queue*. A special object usually called *dispatcher* or *scheduler* which runs in the context of the active object thread dequeues and processes each incoming message and then invokes the actual methods of the servant. Possible results of asynchronous methods are returned in the form of future objects. Figure 2.1 shows the corresponding message sequence chart for a typical request from a client to an active object.

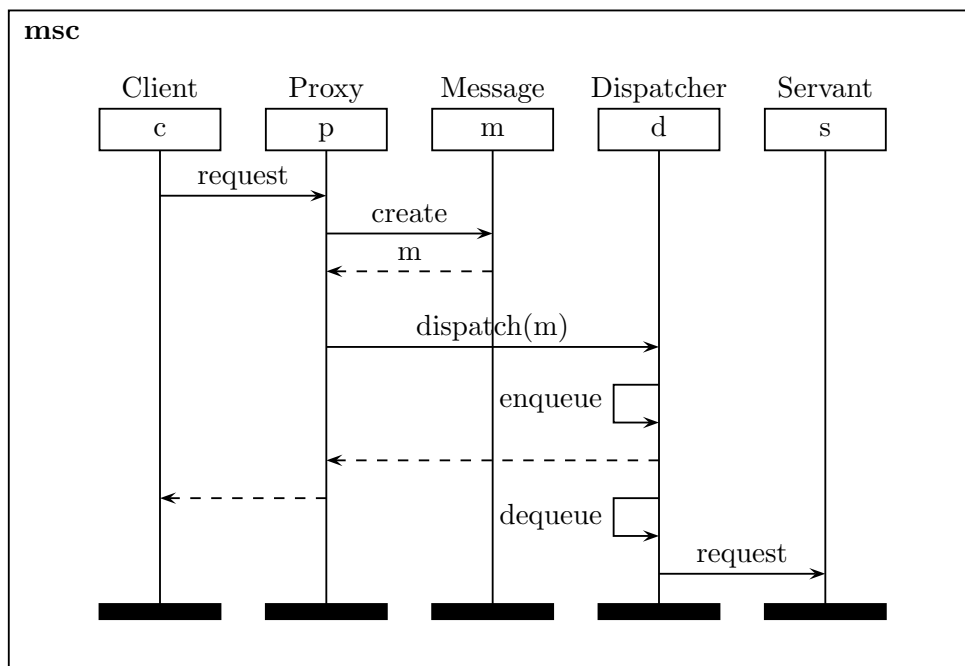


Figure 2.1: Active Object Behavior

We can observe two important things in this chart. First, it should become clear that the client does only interact with the public proxy interface of the active ob-

ject and is not (or does not need to be) aware of the underlying message-passing semantics. Interaction and communication with the servant is achieved indirectly with request arguments and possible future return values. Secondly, we can see that the dequeue and request operations on the lower right of the chart are processed asynchronously by the dispatcher while the client on the left can complete other tasks in the meantime.

In summary, active objects are characterized by the combination of the following three fundamental properties.

Message-based Requests and possible arguments to an active object are converted into messages, forwarded to and eventually executed by the private active object implementation. Result messages are modeled as future objects.

Asynchronous Requests to an active object are executed asynchronously in a private thread of control and only block the caller if the result is requested before it is available. Asynchronous execution of requests is the property which lets active objects add concurrency and multithreading to applications.

Thread-safe Active objects are inherently thread-safe by sequentially dequeuing and processing enqueued requests and always executing them in a single thread of control. The thread-safety of active objects largely removes the need for manual locking and mutual exclusion mechanisms.

2.1.1 Components

As outlined in the preceding section, an active object consists of several components which interact with each other to build the concurrency abstraction as a whole. In this section, I explain the individual components and general structure of an active object in detail.

As with other software models, the following distinction into individual reusable components is not necessarily the only possible way to interpret the Active Object model. In fact, there are other variants known which combine some of the following parts into single components and understand the underlying structure of an active object as a more integrated model (compare [12, p. 9]). All variants have pros and cons but typically share the same core functionality. A more integrated model typically benefits from creating fewer objects and reduces the runtime overhead of an active object but also has the disadvantage of having tightly-coupled and not reusable components. Flexibility in terms of distinct reusable components is thus traded for a slightly better runtime performance.

Proxy The proxy is the part of the Active Object pattern which lets an active object look like any traditional passive object by hiding the internal message-based model and the necessary thread management from the client. It is responsible for converting method requests and arguments from clients into messages and forwarding them to the dispatcher for asynchronous execution. If necessary, it also creates one or more

future objects, attaches them to the internal messages and finally returns them to the caller as placeholders for real method return values and out parameters. The proxy thus does not contain any object behavior or object data by itself but rather acts as an interface between the clients and the private active object implementation.

Proxies are used in the Active Object pattern to provide a strongly-typed interface to clients with normal methods, parameters and return values as expected from classes and objects in modern object-oriented programming languages. The proxy model usually requires more work when manually designing and writing active classes when compared to a simpler loosely-typed public message-based approach but results in a more familiar and less error-prone interface.

Servant The servant represents the actual implementation of an active object. It encapsulates the active object data and defines its behavior and state. Although the servant could also be an active object, it usually is a normal passive object which does not need to be aware of the active object context it is used in. This allows for a broad range of applications where existing passive classes can be turned into active classes simply by providing a wrapping proxy. The proxy adds concurrent method execution and thread-safety even if the underlying servant does not support these properties by itself.

Messages Messages are used as a transport mechanism for method requests initiated by clients. A message contains a particular message type which identifies the related method of the servant. Other context information like method request arguments or future objects are attached to the message by the proxy to make them available to the dispatcher and the servant. Future objects in turn can be seen as an abstract representation for return messages from the active object to the client.

Guards To allow the scenario of implementing certain constraints and rules within an active object, the Active Object model knows the so called *guard* functionality. Each method of an active object can optionally have an associated guard. Such a guard is normally represented by a simple boolean expression and is usually part of the servant. If the guard expression evaluates to true, the associated servant method can safely be invoked and if it returns false, the current state of the active object does not permit calling the related method. A guard can thus be used to control the access to an active object method and influences the processing order of queued messages.

Priorities Methods of active objects can also have an associated *priority*. Like guards, priorities of methods influence the execution order of queued messages. They can be used to specify that some methods of an active object (and the related messages) are considered more important than others and should be given precedence by the dispatcher. By default, all methods of an active object have the same priority. Possible uses of method priorities include modeling an active object which prioritizes

read access over write access, for example. The supported priorities in increasing order are `lowest`, `lower`, `normal`, `higher` and `highest`. The default priority is `normal`.

Dispatcher The dispatcher runs in the context of the active object thread and is the necessary link between the public proxy and the private implementation. The dispatcher serves two purposes. It first provides operations to append new messages to its private message queue. These operations are called by the proxy and run in the context of the client. The other purpose is to dequeue messages in the context of the active object thread and to invoke the related methods of the servant. As indicated in the preceding sections, the dispatcher does not necessarily dequeue messages in chronological order but can optionally use other criteria like priorities or guards to decide which messages to process next. These rules for deciding which message to dequeue next are as follows.

1. A message cannot be dequeued if the related guard, if any, returns false, i.e. it does not allow calling the corresponding method of the servant.
2. From the set of messages whose guards return true, the messages are dequeued in decreasing priority order, i.e. the messages with the highest priority are dequeued first.
3. If two or more messages are candidates to be dequeued and share the same priority, the messages are dequeued in chronological order as enqueued.

When seen from a higher-level perspective, the proxy and the dispatcher form a special version of the classical producer-consumer problem [3]. The proxy acts as the producer and the dispatcher combines the operations and behavior of the shared queue and the consumer.

2.2 Futures

Futures serve two main purposes in the Active Object pattern. The first and more important purpose of futures is to act as placeholders for the actual results of active object methods. The second and probably less frequently used purpose is to provide a client interface to explicitly wait for a servant method to complete. In most cases, this advanced functionality is not directly needed but there are some cases as we will see later where this level of control can be very helpful and allows certain scenarios that would be difficult without.

Futures are a very convenient way to represent the necessary link between the client and the servant. The proxy of an active object is responsible for creating this link. If an active object method is invoked which returns a future, the proxy creates a new future object, attaches it to the internal queue message to make it available to the servant and then returns it to the client. Once a result of a method has been

computed by the servant, the related future is filled. The client can then access this result value later when needed. If the client tries to retrieve the value before the servant method is completed and the future is filled, it is automatically blocked until the requested value is available. This behavior is depicted exemplary in figure 2.2.

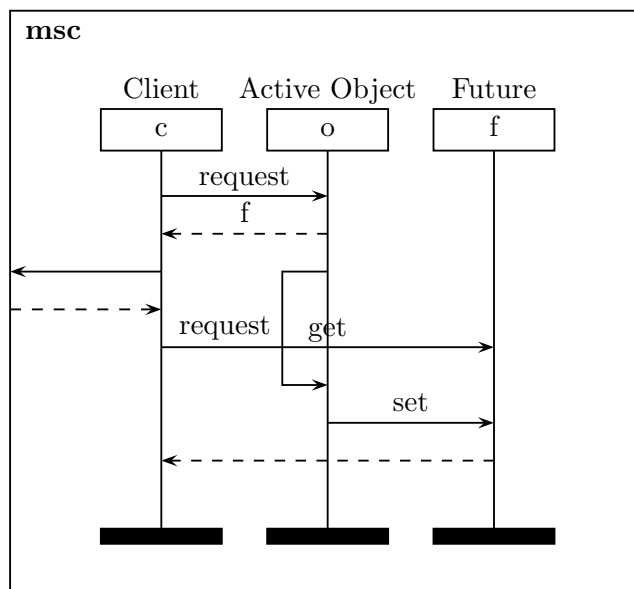


Figure 2.2: Future Blocking Behavior

With this automatic blocking semantics when accessing a request return value or waiting for an active object method, a future can be seen as a simple and loosely-coupled synchronization point between a client and the related active object.

Another name for the future concept is *promise*. Both terms are often used interchangeably.

2.2.1 Errors

Besides the basic functionality of providing access to return values and offering an interface to wait for methods, futures are also responsible for the error handling and error reporting of active objects. Modern object-oriented programming languages and environments like C# and .NET use the notion of exceptions to designate and handle error situation. This is different from other approaches that use normal method or function return values to indicate errors. In this thesis, futures provide explicit exception support and control by automatically rethrowing occurred errors when a client tries to access and retrieve related method results.

2.2.2 States

A future is always in a well-defined *state* after it has been created and returned by a call to an active object proxy method. The following three pre-defined future states are supported.

Pending No value has yet been stored in the future and no error has occurred so far. This is the default state of a future. If a client tries to access a possible future value in this state, this operation blocks until the future changes to one of the following two states.

Succeeded A value has been computed and successfully stored in the related future. Accessing the future value in this state is guaranteed not to block the client and not to throw an exception.

Failed An error has occurred while trying to compute the future value and the related exception object has been stored in the future. Trying to access the future value in this state does not block the caller but results in a rethrown exception.

A future can change its state only once in its entire lifetime. Concrete implementations of a future can provide methods and properties for the client to query the state of a future object as well as methods for waiting for servants and retrieving future values and errors. Also conceivable is a more advanced event based interface which informs interested clients about state changes.

2.2.3 Grouping

Another advanced future related feature is the ability of *grouping* or *composing* futures [24]. A group of futures always consists of exactly two child futures (hereinafter referred to as *left* and *right* children of a future group) and yields a new future which in turn can be used as a child to create further future groups. This way it is possible to build arbitrary complex future hierarchies or *future trees*.

Note that in a future tree, the futures are leaf nodes and the root and intermediate nodes are always represented by future groups. A future tree is a non-empty binary tree [10] with $h + 1 \leq l \leq 2^h$ futures (leaf nodes) and $h \leq g \leq 2^h - 1$ future groups (non-leaf nodes) where h is the height of the tree. These are the minimum and maximum amounts of leaf and root/intermediate nodes in a non-empty binary tree with a given height where every non-leaf node has exactly two children. This is similar to a tree representing an algebraic formula with binary operators only.

The state of a future group depends on the child states. The following two future groups are covered in this thesis.

All The *all future group* stands for the logical AND relation of two futures. The state of this future group changes to succeeded only in the case of two successful

child futures. If one child future has failed for one reason or another, the entire group fails. Table 2.1 lists all possible state combinations.

Left	Right	Group
Pending	Pending	Pending
Succeeded	Pending	Pending
Pending	Succeeded	Pending
Failed	Pending	Failed
Pending	Failed	Failed
Failed	Failed	Failed
Succeeded	Failed	Failed
Failed	Succeeded	Failed
Succeeded	Succeeded	Succeeded

Table 2.1: All Future Group States

Any The *any future group* on the other hand represents the logical OR relation of two futures. The state of the future group changes to succeeded in the case of at least one successful child future. The entire group fails only if both child futures were unsuccessful. Table 2.2 lists all possible state combinations.

Left	Right	Group
Pending	Pending	Pending
Failed	Pending	Pending
Pending	Failed	Pending
Failed	Failed	Failed
Succeeded	Pending	Succeeded
Pending	Succeeded	Succeeded
Succeeded	Failed	Succeeded
Failed	Succeeded	Succeeded
Succeeded	Succeeded	Succeeded

Table 2.2: Any Future Group States

2.3 Active Blocks

Closely related to active objects, futures and future groups is the concept of active blocks. Conceptually, an active block is a block of code statements which can be executed concurrently as a whole (compare figure 2.3). Similar to methods of active objects, active blocks can compute and return values, also encapsulated in futures.

Active blocks cannot take parameters, but can optionally consume one or more local variables. In this case, it is important to note that an active block accesses and modifies the original variable which is declared outside the block and does not refer to a local copy. This is semantically similar to the concept of closures where referenced variables are said to be *bound* to a closure. Unlike closures, active blocks are not treated as functions which need to be called explicitly but are embedded into other code and are executed implicitly just like any other block of statements.

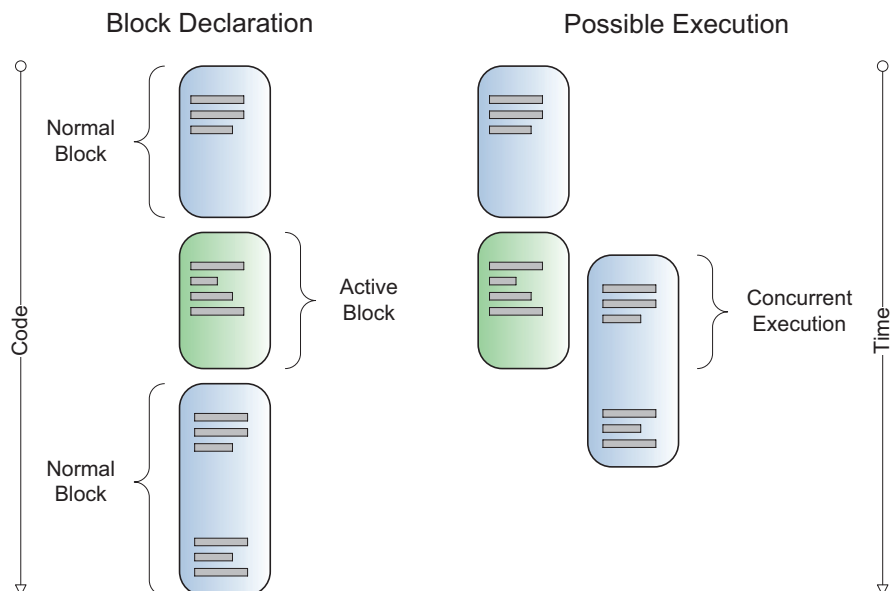


Figure 2.3: Active Block Declaration and Behavior

As we will see later, active blocks are a very convenient means of asynchronously executing certain parts of code without the need to declare and implement dedicated active classes. Chapters 3 and 4 demonstrate how active blocks look like syntactically, why they are useful and how they interact with futures and local variables.

2.4 Related Concepts

In this section, I give examples of other technologies and concepts which share one or more characteristics with the Active Object model. This list is not exhaustive by any means but is meant to indicate that some of the fundamental ideas and principles used in the Active Object model and this thesis can also be found elsewhere.

2.4.1 Remote Objects

The fundamental concept of the Active Object model of using messages for communication and splitting objects into separate proxies and servants can also be found in technologies which are not necessarily related to concurrent programming. For

instance, technologies like Remote Method Invocation [7] or Remoting [4] use similar techniques to provide a seamless and convenient way to communicate with remote objects.

This usually works as follows. On the client side, requests to a proxy or *stub* and its arguments are marshaled into a format which can be transferred across process boundaries and send to a server. This marshaling process is typically called *serialization*. On the server side, the transferred data is received by the *skeleton*, then *deserialized* again and finally passed to the actual remote object. Return values are handled the other way round, i.e. first serialized by the skeleton and then deserialized by the stub. The remote communication as well as the serialization and deserialization tasks are performed transparently and are completely hidden from the client and the remote object.

Despite the similarities between remoting frameworks and the Active Object model, there are also a few differences. First, remoting frameworks are designed to work with objects on different physical machines or at least in different operating system processes whereas the Active Object model as presented in this thesis only uses separate threads of control for the proxies and servants. This has a huge impact on the performance since remoting frameworks are forced to use slower inter-process communication methods [21] such as network sockets [20], pipes or shared memory for transferring data. Secondly, remoting frameworks usually block the client until the remote object has processed the request. This is contrary to the asynchronous Active Object model.

There is also a version of the Active Object model known as *distributed active object* [12, p. 10] which combines the functionality of traditional active objects with those typically found in remoting frameworks by processing requests both remotely *and* asynchronously.

2.4.2 Active Monitors

Also closely related to this thesis is the work of Andrews in the form of active monitors [1]. Active monitors are very similar to active objects in that both abstractions use messages for communication and active processes (threads) to introduce concurrency as well as guarantee implicit mutual exclusion by sequentially dequeuing and processing incoming requests. Request channels of active monitors are message queues in the terminology of active objects and replies and results are modeled with futures.

One difference, however, is the fact that active objects provide a strongly-typed interface to clients via ordinary method calls and parameters whereas active monitors rely on message types or multiple channels to differentiate between requests.

Chapter 3

Design of Implementation

After introducing the basic concepts of the Active Object model in general, in this chapter I give a summary of the developed active object runtime library and the active object compiler. I also point out some of the difficulties and challenges I experienced during the implementation. At the end of this chapter, I discuss the required minimum features to provide a comparable concurrency abstraction in programming languages other than C[#] and also compare C[#] and .NET with other object-oriented languages and environments such as C++ and Java.

3.1 Runtime

The first part of the practical aspects of this thesis is the active object runtime library. It implements the basic concepts of the Active Object model from chapter 2. The runtime library consists of several interfaces, classes and enumerations and is implemented as a .NET class library forming a single .NET assembly. This section is dedicated to explaining the structure, behavior and implementation of the key components of the runtime library in detail. Note that not all components of the runtime library are listed here. Helper classes and other less important types are omitted.

3.1.1 Dispatcher

The `Dispatcher` class and its base interface `IDispatcher` are responsible for enqueueing, dequeuing and dispatching incoming messages from proxies and the entire thread and synchronization management of an active object. Each active object creates and manages its own dedicated `Dispatcher` object.

The concrete implementation of the `Dispatcher` class closely follows the behavior of the abstract dispatcher component as described in section 2.1. Once a new method request has been passed to the `Dispatch` method (refer to the abstract class diagram in figure 3.1), it is enqueued and scheduled for asynchronous execution. Control is immediately returned to the caller. Concurrently, the `Dispatcher` dequeues and

processes enqueued requests in an internal thread and invokes the related methods of the servant. The queue functionality thereby is implemented as a priority-based queue [2] with additional support for guards and follows the same rules as in section 2.1.1.

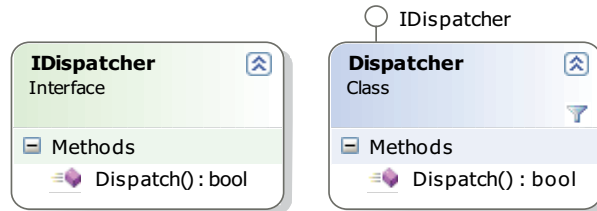


Figure 3.1: Dispatcher Class Diagram

As depicted in figure 3.1, the `IDispatcher` interface does not provide any methods for explicitly starting and stopping the internal active object thread. The entire thread and synchronization management including thread creation and shutdown is handled transparently by the `IDispatcher` interface and is hidden from the proxy of an active object. The `IDispatcher` interface only provides a single operation for dispatching new method requests from clients. This transparent thread management removes the need for explicit thread shutdowns by clients and greatly simplifies the usage of the active object runtime library but also turned out to be quite a challenge to implement for the following reasons.

Due to its garbage-collecting memory management, .NET and C# do not support deterministic object destructions. Possible destructors or *finalizers* [19, p. 40] are eventually called by the garbage collector but it is not specified or guaranteed when this will happen. This is contrary to other languages such as C++, where objects are destroyed deterministically [22, pp. 244-257]. Theoretically, it would have been acceptable that an active object cleans up and stops its thread only when eventually called by the garbage collector. But the problem in the particular case of .NET is, that .NET does not allow to reference other managed objects in object finalizers. This makes it unfortunately impossible to use a combination of the garbage collector and object finalizers to stop and cleanup active objects threads.

To overcome this shortcoming, .NET knows the concept of *disposable objects* in the form of the `System.IDisposable` interface. This interface offers a single method called `Dispose` which can be seen as a way to deterministically cleanup objects. One solution could thus have been to inherit proxies from `System.IDisposable`, implement `Dispose` in a way to stop the dispatcher and to let clients call this method when an active object is no longer needed. While this solution sounded reasonable at first, I decided against it because I did not want to force clients to call a cleanup routine explicitly for each active object. After all, one major benefit of the .NET platform is the garbage collector and the goal was not to change the implicit memory management and cleanup semantics for an active object if the related servant does

not require this.

I eventually solved this problem by creating and stopping active object threads dynamically on demand. The concrete implementation of the `Dispatch` method ensures that there is always exactly one internal dispatcher thread active at the same time. If the dispatcher queue is empty and the internal thread does not have any more items to process, this thread exits automatically and another thread is used for the next method request(s). To avoid possible performance issues by creating new threads over and over again, I decided to use a thread pool for recycling existing threads and minimizing the thread creation overhead.

3.1.2 Future

The `Future` base class represents a future of the Active Object model. As shown in figure 3.2, it provides methods and properties for querying its current state as well as for waiting for active object methods and active blocks and has built-in support for error reporting and exception handling. A `Future` object can optionally be associated with a result value of an active method or active block in the form of `Future<T>`, a generic [18, pp. 52-56] version of `Future`. Accessing the value of a `Future<T>` object follows the same semantics as in section 2.2.2.

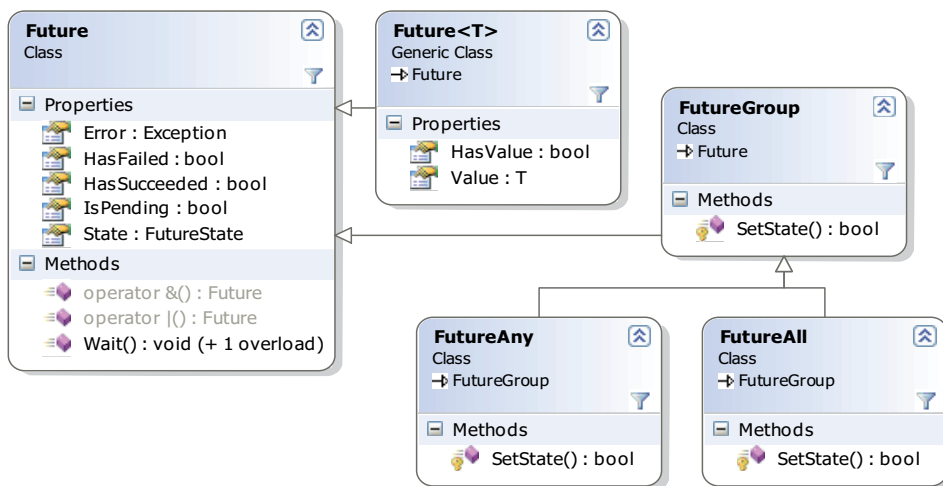


Figure 3.2: Future Class Diagram

Future groups as described in section 2.2.3 are also part of the runtime library and are modeled with the `FutureAll` and `FutureAny` classes and by overloading the logical language operators `&` and `|` in the `Future` class, respectively. The base class for the future groups, `FutureGroup`, thereby implements most of the functionality and the derived classes are only responsible for the individual future group state combinations. Future groups support the same properties and operations as ordinary futures.

The implementation of `Future` and its descendants heavily depends on the functionality of the `System.Threading.Monitor` class, the equivalent of the monitor [8] concept for .NET. Its lock features are used for ensuring thread-safety for otherwise non-atomic operations and the condition variables provide us with the necessary signal and wait behavior. Correctly implementing the signal and wait behavior with condition variables turned out to be a bit tricky since monitors do not maintain state when signaled. This behavior can lead to situations where lost signals result in deadlocking waiting clients. I solved this problem by checking the future state before issuing a wait call. Although .NET offers alternatives for implementing the signal and wait behavior which do maintain state such as event handles, I opted for the monitor solution because these alternatives require explicit cleanups, something that monitors do not.

3.1.3 Operation

The `Operation` class and its base interface `IOperation` reflect the idea of messages and method requests of the Active Object pattern. `IOperation` objects are used for invoking methods of the underlying servant and for transferring method request arguments and possible `Future` objects to the internal active object thread. An `IOperation` object is created and dispatched for each invoked method of the proxy interface. It has an associated priority and can optionally specify a guard to control the access to itself and the related servant method. Figure 3.3 shows the corresponding class diagram.

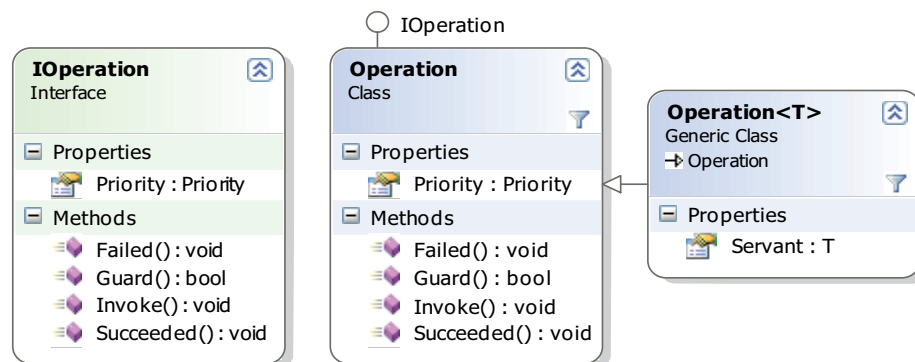


Figure 3.3: Operation Class Diagram

Note that the proxy of an active object does not create instances of the `Operation` class directly but rather of `Operation` subclasses. `Operation` only provides a basis by implementing the `IOperation` interface with the default behavior, i.e. it has a priority of `normal`, specifies no guard and does not call any servant method when invoked by the dispatcher. It is a common pattern that the proxy defines an `Operation` subclass for each offered class and instance method and then creates and dispatches objects of these classes instead of using the general `Operation` type. The `Invoke`

methods of these custom classes are then responsible for calling the related methods of the servant. In addition to the methods and properties defined by the `IOperation` interface, the subclasses normally also provide strongly-typed properties or fields for storing method request arguments and possible `Future` objects.

3.1.4 Tasks

The `Tasks` class is an interface for managing active blocks. Its primary purpose is to provide static [19, pp. 23] methods for scheduling one or more active blocks for asynchronous execution as well as for waiting for groups of active blocks. Active blocks themselves are modeled with the help of delegates [18, p. 45] and especially anonymous methods [18, pp. 56-59], a language feature of `C#` which is comparable to the concept of closures. Figure 3.4 shows the corresponding class diagram. Section 3.3.3 lists several examples on how the `Tasks` class can be used to emulate active blocks with `.NET` and `C#`.

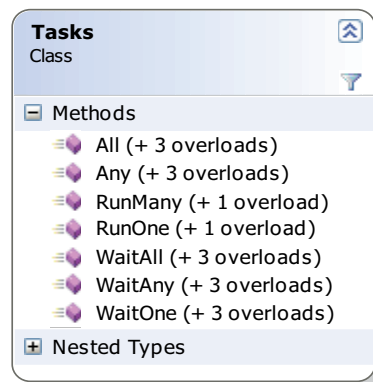


Figure 3.4: Tasks Class Diagram

The core implementation of the `Tasks` class is very similar to the behavior of the `Dispatcher` type. Internally, the passed active blocks or, to be more precise, the delegates are wrapped into custom `Operation` objects and then scheduled for asynchronous execution. The threading behavior thereby is also implemented with the help of the thread pool.

3.2 Compiler

The active object compiler is responsible for auto-generating the public proxy interface code of an active object. It leverages the available reflection [14] and metadata capabilities of `C#` and `.NET` in the form of custom language attributes [18, pp. 51-52] to provide a very convenient way to declare classes as active classes. The active object compiler thus significantly reduces the effort involved in working with active

objects by avoiding the repetitive tasks of writing the required proxy interfaces by hand.

This works as follows. The compiler takes an annotated class definition of the active object servant as input (compare section 3.3.1) and outputs the corresponding proxy interface (see figure 3.5). The input is expected to be given in compiled form and the proxy is output as C# source code. Due to the nature of the .NET platform and the input format, the frontend of the compiler can work with any .NET compatible language. The backend is currently limited to emitting C# source code. It can, however, be enhanced to emit other output formats such as source code for programming languages other than C# or even pre-compiled .NET classes.



Figure 3.5: Active Object Compiler Structure

3.3 Examples

3.3.1 Active Objects

One of the active object examples used throughout this chapter and the rest of this paper deals with a `Buffer` class, a thread-safe queue with asynchronous operations for enqueueing and dequeuing items. This section introduces the `Buffer` class to explain the basic procedure of implementing active classes and then proceeds step-by-step to more advanced features like priorities and guards.

Listing 3.1 shows the annotated servant of the active class which provides the core functionality of the queue. It basically offers two simple operations, `Put` and `Take`, for appending a new item to and removing an existing item from the internal queue.

```

1  [Active]
2  class Buffer {
3      private int m_Capacity;
4      private Queue m_Queue;
5
6      public Buffer(int capacity) {
7          m_Capacity = capacity;
8          m_Queue = new Queue();
9      }
10
11     [Priority(Priority.Normal)]
12     public void Put(object item) {
13         m_Queue.Enqueue(item);
14     }
  
```

```
15
16     [Guard("Put")]
17     public bool PutGuard() {
18         return m_Queue.Count < m_Capacity;
19     }
20
21     [Priority(Priority.Higher)]
22     public object Take() {
23         return m_Queue.Dequeue();
24     }
25
26     [Guard("Take")]
27     public bool TakeGuard() {
28         return m_Queue.Count > 0;
29     }
30 }
```

Listing 3.1: Annotated Buffer Class

The active object compiler enables us to combine the public proxy interface and the private servant of an active object into a single annotated class declaration. The compiler is responsible for analyzing such a class and emitting the corresponding proxy interface. An active class is identified by the `Active` attribute, guards use `Guard` and priorities are modeled with `Priority`. The generated proxy has the same interface as the servant, except for the return values of `Put` and `Take` replaced with futures. `Put` returns a value-less non-generic future whereas `Take` returns a generic future of type `object`. Upon creation, the proxy creates an instance of the servant and stores a reference to it in a private field. For each of both public methods, `Take` and `Put`, it defines an inner operation class which is responsible for invoking the related method of the servant. When triggered by a client call, `Put` and `Take` create an instance of the related operation class, attach the servant and possible arguments as well as future objects and finally schedule this operation object for asynchronous execution with the help of the dispatcher.

Guards `Buffer` would not be complete without defining and specifying client rules. For example, it should not be possible for clients to invoke the `Take` operation of the servant if the underlying queue does not contain any items. Likewise, calling `Put` should be prevented if the queue is currently considered full. This behavior and more can be implemented by adding guards.

For this purpose, the servant defines two methods `PutGuard` and `TakeGuard` which implement the internal constraints and are annotated with the `Guard` attribute. The `Guard` attribute expects the name of the class member to protect. Implementation-wise, these guard methods are then used by the generated proxy for the related operation classes.

It is important to note that calls to `Put` and `Take` of the proxy do not block the caller even if the related guards currently evaluate to false. Calls to an active object method are always enqueued, return immediately and are eventually executed

asynchronously in the thread context of the active object. This is different from conditional waiting with condition variables where unsatisfied conditions always result in a wait operation for the caller. This conditional waiting can be emulated in the Active Object model, if desired, by waiting on returned future objects. For instance, a producer could occasionally wait on the returned future of `Take` in order to guarantee that even slow consumers can handle the amount of generated items.

Priorities As outlined in section 2.1.1, priorities can influence the execution order of queued messages. We use this functionality in our `Buffer` example to specify that `Take` should be given precedence over `Put` by the dispatcher and therefore prioritize client read access over write access. Implementation-wise, priorities are added by changing the priority property of the operation objects in the generated proxy methods.

The concrete example uses only two priorities, `normal` and `higher`, to specify the different operation order semantics. More complex scenarios can be modeled by also incorporating the other priorities.

3.3.2 Future Groups

As outlined in section 3.1.2, future groups and future trees are supported by the runtime library by overloading logical C[#] language operators. The all future group uses the logical AND symbol `&` and the any group is associated with the logical OR symbol `|`. Implementation-wise, creating a future group yields a new future which in turn can be used to create other future groups. This is demonstrated in the last line of listing 3.2 which first creates an any future group of `b` and `c` which is then combined with another future `a` to form an all group.

```
1 Future f = b | c; /* b or c */
2
3 ...
4
5 if (f.IsPending) {
6     ...
7     f.Wait();
8 }
9
10 ...
11
12 (a & (b | c)).Wait(); /* Composing groups */
```

Listing 3.2: Future Groups

Figure 3.6 shows the corresponding future tree for `a`, `b` and `c`. Recall from section 2.2.3 that the amount of possible nodes in a future tree with a given height is well-defined. In this example, we have the minimum possible amount of nodes for a future tree of height $h = 2$ with $l = h + 1 = 3$ futures and $g = h = 2$ future groups.

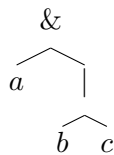


Figure 3.6: Example Future Tree

3.3.3 Active Blocks

This section is dedicated to demonstrating how active blocks can be used with the runtime library. I present two basic examples which are meant to give a good overview of how active blocks look like syntactically and why they are useful at all.

I begin with a minimal example in listing 3.3 which asynchronously sorts a local array. The actual sorting is implemented by an anonymous method which references the local array of the enclosing block. The resulting delegate is passed to the `Tasks` class for asynchronous execution and the returned `Future` object is saved for later access. The call to the `Wait` operation in the last line of the example acts as a synchronization point between the active block and the enclosing statements and is issued before the local array is needed again.

```

1 Future f = Tasks.RunOne(
2     delegate { Array.Sort(s); }
3 );
4
5 /* Do some other work .. */
6 ...
7
8 f.Wait(); /* When the array is needed again */
  
```

Listing 3.3: Sorting an Array

It is important to note that it would not be considered thread-safe if the code section between scheduling the active block and the corresponding future `Wait` call modified the same local array. Since sorting an array is not thread-safe by itself, doing so could result in an unpredictable behavior such as an unsorted array after the `Wait` call had returned. In general, care must be taken that local variables that are used simultaneously in active blocks are accessed or modified by the enclosing statements only if the multithreaded environment is taken into account. That means, operations on such local variables must either be thread-safe or may only be applied outside the critical regions between the active block and possible synchronization points.

Listing 3.4 is very similar to the sort example above but differs in that the active block returns a generic future this time. This example searches a local array and the generic future is a placeholder for the array index to find. Once this index is needed, the example issues a call to the `Value` property of the returned future. This is demonstrated in the last line of the listing. The call to `Value` may or may not

block depending on the fact whether the active block has already found the index or not at this time. In the former case, the index is returned immediately and in the latter case, the call blocks until the active block has processed the search request.

```
1 Future<int> f = Tasks.RunOne<int>(  
2     delegate { return Array.IndexOf<int>(s, 100); }  
3 );  
4  
5 /* Do some other work .. */  
6 ...  
7  
8 int index = f.Value; /* When the index is needed */
```

Listing 3.4: Searching an Array

3.4 Requirements

Implementing the Active Object model for a particular object-oriented programming language and environment requires a minimum set of available language features and components. The following list describes the language features and components which I found to be especially important or useful. Note that some of the following points are not strict requirements but rather recommendations which can simplify the development or usage of an active object runtime library. The points are listed in decreasing significance and importance.

Threads A basic API for creating, stopping and managing threads is essential for implementing the Active Object model. The entire Active Object model as presented in this thesis totally depends on the ability to create custom threads at runtime. Other ways of introducing concurrency are conceivable (compare section 2.4) but are not elaborated in this thesis. Also needed besides a threading API are synchronization primitives such as semaphores [3] or monitors to protect critical regions of code from being entered by multiple threads at the same time and to implement the signal and wait behavior as provided by futures.

Generics The ability to create and consume generic types is a requirement for being able to implement the abstraction of a *polymorphic future* [12, p. 10] (also compare section 3.1.2). Polymorphic futures are needed to provide a strongly-typed interface with minimal programming effort for accessing return values of active methods or active blocks. For languages without support for generics, polymorphic futures can be emulated to some extent by using a more general type for the value of a future or by defining multiple future classes each associated with a particular type. Both workarounds are far from ideal but can be used if generics are not available.

Closures Closures are ideal for emulating active blocks in a programming language. Both concepts share similar semantics like their ability to reference variables

declared in outer blocks and the general structure of grouping statements (compare sections 2.3 and 3.1.4). Being able to consume and modify local variables in active blocks turned out to be one of the key features in the entire concept of active blocks since this enables application developers to asynchronously process certain tasks with minimal programming effort. For languages without native support for closures, active blocks can still be implemented but then usually lack the important feature of being able to conveniently reference other variables.

Garbage Collection A garbage collector is not strictly a requirement per se, but can greatly simplify the usage of an active object runtime library by removing the burden of manually freeing allocated objects. The Active Object model as presented in this thesis can generate a lot of short-living objects whose ownership change over time. Moreover, using future groups can involve creating several intermediate objects which are only referenced internally while evaluating the generated future tree and the group state and usually not by the client itself. Both facts make it difficult to agree on a reliable memory deallocation model.

Techniques like reference-counting can help to some extent if a particular language does not support garbage collection, but a garbage collector is the preferred way to handle the memory management in the Active Object model.

Operator Overloading Overloading language operators is used for providing a natural and intuitive way to build future trees (compare section 3.3.2). Similar to the aforementioned garbage collector, this feature is not strictly necessary but rather nice to have. Overloaded language operators can actually be replaced with ordinary methods but usually result in cleaner and easier to read code.

3.5 Comparison

After introducing some general requirements and feature recommendations to implement a comparable active object runtime library in an object-oriented programming language other than C#, I use this section to examine some programming languages in detail. Included in this discussion are C++ and Java. Note that the requirements for an active object compiler are not taken into account in this section since its implementation is quite specific to the .NET platform and C#. This section thus only discusses the runtime library itself.

3.5.1 Java

The term Java refers to both the *Java language* [6] and the related *Java platform*. The Java platform and runtime environment provides a virtual machine to execute applications which are compiled to so called *intermediate code* or *byte code*. This byte code is machine-independent and can thus be used to run Java applications on a

variety of platforms. The Java language on the other hand is an object-oriented programming language whose compiler emits Java byte code. Comparing this concept to the .NET world, we can say that the Java language can be seen as the C# counterpart whereas the Java platform is the equivalent of the .NET runtime environment.

The Java language and platform are in many aspects very similar to C# and .NET. They share the same core concepts and provide a similar feature set. That's why it comes with no surprise that the Active Object model can be implemented in Java in nearly the same way as in C# and .NET as I explain in this section.

The most important requirement for implementing the Active Object model, the threading API, is available in the form of the `java.lang.Thread` class and the related `java.lang.Runnable` interface. Both types are responsible for starting and stopping custom threads at runtime and are thus suited to add concurrency to Java applications. Moreover, the other threading related requirement, synchronization primitives, is supported by Java with its native support for monitors. Monitors are implemented with a combination of ordinary method calls and built-in language support: monitor locks can be applied with `synchronized` blocks or methods and the `wait` and `notify/notifyAll` methods of the `java.lang.Object` class represent the interface for condition variables. In newer Java versions, the `java.util.concurrent` package provides additional concurrency features such as thread pools and more advanced and feature rich lock implementations as well as other synchronization primitives such as semaphores.

The next point on the list, the polymorphic future type, can be modeled in newer versions of Java just like in my example runtime library for C# and .NET. Since version 1.5, Java provides native support for generics and is thus suited to declare and consume generic future types. No workarounds are necessary to provide a comparable abstraction in Java.

Closures, however, are only partially supported by Java. Partially because Java knows the concepts of so called local and anonymous classes which share a few but not all properties of closures. Local classes, as the name implies, can be declared locally within a member of a class such as a method or a constructor. Local classes can access fields and methods of the enclosing class as well as local variables and parameters of the enclosing member. Anonymous classes in turn are basically local classes without a name. Being able to consume local variables in local classes is very similar to binding variables to a closure. One minor but important difference thereby lies in the fact that Java can only consume but not modify local variables. Java only supports referencing *final*, i.e. read-only, local variables within local classes. However, if needed, this can be worked around by using member fields of the enclosing class instead of local variables in local classes. Another difference between closures and local classes is that local classes are in fact real full-blown classes whereas closures behave and are usually declared more like an ordinary function.

The next recommendation is the garbage collector. Just like in C# and .NET, the memory management in Java is not handled manually but covered by a garbage collector. This results in a simple object allocation and deallocation model but unfortunately also involves the same object destruction implications as described in

section 3.1.3. Java follows the same object destruction semantics as C[#] and .NET in that possible object finalizers are not called deterministically but eventually run by the garbage collector. This scenario can be handled in Java in a similar way to my aforementioned dynamic thread management and thread pool solution for C[#].

The last point, custom operator overloading for building groups of futures, is not yet available for Java and would need to be replaced with ordinary methods.

3.5.2 C++

C++ [22] is an object-oriented programming language invented by Bjarne Stroustrup in the early 1980s and is based on the traditional procedural programming language C [9] from Dennis Ritchie. C++ differs from .NET and C[#] in that C++ applications usually do not run within the boundaries of a virtual machine but are executed natively on the respective platforms. The feature set of both programming languages and environments with respect to the Active Object model are comparable but differ in some points as I explain in this section.

In contrast to C[#] and .NET, C++ does not yet provide any kind of standardized threading API. However, there are several ways to work around this. First, it is always possible to use a threading API of the underlying operating system, such as the threading API from Microsoft Windows. Secondly, it is also conceivable to use a third-party C or C++ based threading API as provided, for instance, by the POSIX Threads library. Whatever option is chosen, these APIs or libraries provide the necessary functionality to create and destroy new threads, to synchronize sections of code and to block and signal threads.

The next point on the list, the polymorphic future type, can easily be implemented by means of C++ templates [22, pp. 327-354] (also compare [12, p. 10]). C++ templates provide the necessary functionality to create and consume generic future types. Thus, the future types can be implemented in C++ in a very similar way to C[#] and .NET.

Closures, however, are not yet supported by C++. This makes it somewhat difficult to express active blocks in C++ in a convenient way. But since a closure is actually nothing more than another way to group code and data, they can be emulated in object-oriented programming languages with traditional classes and instances thereof (compare function objects [22, pp. 287-288], for instance). This is far from optimal in practice but at least one theoretical solution to model active blocks in C++ today.

The next recommendation is the garbage collector. Unlike C[#] and .NET, the memory management in C++ is handled manually and is not covered by a garbage collector. In order to being able to use a simple memory management model in C++ none the less, one can leverage C++ supported techniques such as reference-counting and smart pointers. These options are not optimal in my opinion but still a significant improvement over a complete manual approach. They even have a small advantage over a garbage-collected solution: reference-counting and smart pointers have the property of deterministic object destruction, something that C[#] does not provide au-

tomatically as explained previously. Thus, a possible implementation of the Active Object model in C++ does not necessarily require any complicated dynamic thread management solutions.

The last point, custom operator overloading for building groups of futures, is fully supported by C++. No workarounds are necessary to provide a comparable abstraction in C++.

Chapter 4

Proposal for C[#] Extension

In addition to presenting the theoretical aspects of the Active Object model as well as developing the runtime library and the active object compiler, I also propose an extension for the C[#] language itself which makes active objects, futures and active blocks first class language constructs. This chapter explains how such an extension can look like and discusses some of the advantages and disadvantages of integrating the Active Object model directly into the programming language.

4.1 Concepts

The basic idea of supporting active objects, futures and active blocks directly in the programming language seems to be natural. Especially active objects and active blocks map closely to several well-known concepts and abstractions of modern programming languages. Classes of active objects are in fact only normal classes with a slightly different behavior and can be modeled, declared and used in a similar way. Moreover, active blocks are actually the same as normal statement blocks, except that they are executed asynchronously and can optionally compute and return result values.

The ideas presented in this chapter are influenced by and based on the work of [24] for C++. A similar language extension for C[#] introduces several new keywords and ideas, some of which can be found in the example listings of this section. A complete language extension reference is given at the end of the following respective sections. An underlying implementation of the language extensions may be based on the active object runtime library and would replace the functionality of the active object compiler.

4.1.1 Active Objects

The first C[#] language extension example depicted in listing 4.1 reuses the `Buffer` example from the previous chapter and replaces the characteristics of the runtime library and active object compiler with the corresponding C[#] language support and

keywords. `Buffer` is now identified as active class by specifying the `active` keyword in the class declaration. Moreover, the guards have changed from dedicated methods to simpler inline `guard` blocks. Priorities are still modeled as custom language attributes.

```

1  active class Buffer {
2      private int m_Capacity;
3      private Queue m_Queue;
4
5      public Buffer(int capacity) {
6          m_Capacity = capacity;
7          m_Queue = new Queue();
8      }
9
10     [Priority(Priority.Normal)]
11     public void Put(object item) {
12         guard (m_Queue.Count < m_Capacity) {
13             m_Queue.Enqueue(item);
14         }
15     }
16
17     [Priority(Priority.Higher)]
18     public object Take() {
19         guard (m_Queue.Count > 0) {
20             return m_Queue.Dequeue();
21         }
22     }
23 }

```

Listing 4.1: Buffer with C[#] Support

active The `active` keyword serves two purposes. First, it is used to declare classes as active classes. When found at the beginning of a class declaration, the `active` keyword is interpreted as a so called *class modifier* and instructs the compiler to treat a class as an active class. Syntactically, the following grammar changes are sufficient for this purpose.

$$\langle \text{class-modifier} \rangle \rightarrow \dots \mid \text{active} \mid \dots$$

Semantically, an active class is handled by the compiler in exactly the same way as a manual active class implementation with the explicit separation into the public proxy and the private servant. The class which is declared as active class is taken as the servant and the compiler is responsible for generating the corresponding public proxy interface. It is important to note that possible clients do not create instances of the servant but rather of the resulting proxy. This is achieved by giving the proxy class the original name of the servant and renaming the servant. The generation of the proxy interface thereby follows the following rules.

1. For each public method or property of the servant, the compiler generates a private inner class in the proxy which calls the corresponding member of the servant and provides public fields for storing possible arguments and return values.
2. For each public constructor of the servant, the compiler generates the same public constructor for the proxy. The implementation of these proxy constructors thereby creates an instance of the related servant and simply forwards the supplied arguments to the constructors of the servant. A reference to the resulting servant object is stored in a private field of the proxy. Moreover, the proxy constructors are responsible for creating an instance of the `Dispatcher` class and storing a reference to it in a private field. If the servant does not define any explicit constructors, the compiler generates a default constructor with no arguments which also creates an instance of the servant and a private dispatcher. Static active classes are handled a bit differently in that no constructors are generated and no servant instances are created.
3. For each public method or property of the servant, the compiler generates a public method or property in the proxy with the same name and signature as in the servant but with possible return values and out parameters replaced with the corresponding generic future type. Members which do not have a return value are modeled to return a value-less non-generic future. The generation of these proxy members thereby follows the same rules as a manual implementation, i.e. they create an instance of the related operation class, attach the servant instance if available as well as possible arguments and future objects and schedule this operation instance for asynchronous execution with the help of the private dispatcher. Member priorities may be integrated into the proxy by passing the desired priority to the related operation instance.

Inheritance with non-static and non-sealed active classes is permitted as long as an active class only derives from an active class and a passive class only inherits from a passive class. Mixed inheritance such as deriving an active class from a passive class is not allowed in order to being able to guarantee the active object semantics for all class members.

Not allowed in an active class are public fields since they can break the thread-safety property of an active object. They would need to be replaced with ordinary properties or methods.

guard A member of an active class can be protected with an embedded *guard statement*. Syntactically, a guard statement is a normal block with a related boolean expression enclosed in parentheses and is identified by the `guard` keyword. In grammar notation, this looks as follows.

$$\begin{aligned} \langle \text{embedded-statement} \rangle &\rightarrow \dots \mid \langle \text{guard-statement} \rangle \mid \dots \\ \langle \text{guard-statement} \rangle &\rightarrow \text{guard} (\langle \text{boolean-expression} \rangle) \{ \langle \text{statement-list} \rangle \} \end{aligned}$$

A guard statement can only appear inside a method or property of an active class and must then be the first statement. In a similar way to a manual active class implementation, such a guard statement or, to be more precise, the related boolean expression is transformed by the compiler into a dedicated guard method in the servant and integrated into the corresponding inner operation class. The remaining statements inside the guard block then form the body of the original class member.

4.1.2 Futures

Minimal C[#] language support is proposed for futures themselves. Futures are covered in the form of a new `future` keyword which can be seen as an alias for and used in place of the `Future` type. This is similar to the built-in type aliases of C[#], like `int` for `System.Int32` [16]. Both normal and generic futures can be supported with the proposed syntax. An example of how this can look like can be found in listing 4.2. No explicit C[#] compiler support is needed for building future groups since this is already covered by overloading language operators (compare listing 3.2).

Besides the `future` keyword itself, this section introduces several additional keywords and statements which are related to the concept of futures.

```

1  future<object> f = m_Buffer.Take();
2
3  ...
4
5  object item = f.Value;
```

Listing 4.2: `future` Aliases

future As outlined in the introduction of this section, the `future` keyword is merely an alias for the `Future` type. The generic version of `Future`, `Future<T>`, is represented by `future<T>`. Syntactically, the following future related grammar change are sufficient. The last two rules thereby are only used in subsequent paragraphs of this section and are not directly related to the `future` keyword itself.

$$\begin{aligned} \langle \text{predefined-type} \rangle &\rightarrow \dots \mid \text{future} \mid \text{future}\langle T \rangle \mid \dots \\ \langle \text{future-expression-list} \rangle &\rightarrow \langle \text{future-expression} \rangle \mid \langle \text{future-expression-list} \rangle , \\ &\langle \text{future-expression} \rangle \\ \langle \text{future-expression} \rangle &\rightarrow \langle \text{expression} \rangle \end{aligned}$$

all The `all` keyword marks the beginning of a so called *all block*. An `all` block expects a non-empty list of future expressions and returns a new future object which is the combined all group of the evaluated futures. Syntactically, an `all` block is identified by the `all` keyword, followed by a pair of curly braces enclosing a comma-separated list of future expressions. In grammar notation, this looks as follows.

$$\begin{aligned} \langle \text{embedded-statement} \rangle &\rightarrow \dots \mid \langle \text{all-statement} \rangle \mid \dots \\ \langle \text{all-statement} \rangle &\rightarrow \text{all} \{ \langle \text{future-expression-list} \rangle \} \end{aligned}$$

Semantically, the passed future expression list must evaluate to future expressions of the same constructed generic type [18, p. 387] or to a list of non-generic value-less future expressions. A mixed list between non-generic and generic futures or a list with different constructed generic future types is not allowed.

The returned future can be used to wait for the combined all group of the passed futures. A concrete implementation of the all block can look as follows. Assuming we have a version of the `and` operation which takes two futures and returns the combined all future group for the two passed futures, the pseudo-code in listing 4.3 is a possible implementation of the all operation. It builds an *all future tree* of the passed future list, i.e. a minimum future tree with only all future groups. If the supplied future list has n futures, the resulting tree is of height $h = n - 1$ and has h all future groups. Thus, the all operation is a $\Theta(n)$ time and $\Theta(n)$ space operation.

If an all block is passed a single future expression only, it may safely be omitted by the compiler for optimization purposes as long as the future expression is still evaluated.

```

1  all(futures):
2      f <= null
3      foreach g in futures
4          f <= and(f, g)
5      return f

```

Listing 4.3: `all` Behavior

any The `any` keyword marks the beginning of a so called *any block*. An any block expects a non-empty list of future expressions and returns a new future object which is the combined any group of the evaluated futures. Syntactically, an any block is identified by the `any` keyword, followed by a pair of curly braces enclosing a comma-separated list of future expressions. In grammar notation, this looks as follows.

$$\langle \text{embedded-statement} \rangle \rightarrow \dots \mid \langle \text{any-statement} \rangle \mid \dots$$

$$\langle \text{any-statement} \rangle \rightarrow \text{any } \{ \langle \text{future-expression-list} \rangle \}$$

Just like with an all block, the passed future expression list must evaluate to future expressions of the same constructed generic type [18, p. 387] or to a list of non-generic value-less future expressions. A mixed list between non-generic and generic futures or a list with different constructed generic future types is not allowed.

The returned future can be used to wait for the combined any group of the passed futures. A concrete implementation of the any block can look as follows. Assuming we have a version of the `or` operation which takes two futures and returns the combined any future group for the two passed futures, the pseudo-code in listing 4.4 is a possible implementation of the any operation. It builds an *any future tree* of the passed future list, i.e. a minimum future tree with only any future groups. If the supplied future list has n futures, the resulting tree is of height $h = n - 1$ and has h any future groups. Thus, the any operation is a $\Theta(n)$ time and $\Theta(n)$ space operation.

Similar to an all block, if an any block is passed a single future expression only, it may safely be omitted by the compiler for optimization purposes as long as the future expression is still evaluated.

```

1 any(futures):
2     f <= null
3     foreach g in futures
4         f <= or(f, g)
5     return f

```

Listing 4.4: `any` Behavior

waitone The `waitone` keyword initiates a so called *waitone block*. Such a waitone block expects a single future expression and waits on the evaluated future. The return value of the waitone block depends on the fact whether this future wait operation was successful or not. In the former case, the return value is the evaluated future of the passed future expression and in the latter case, the return value is `null`. Syntactically, a waitone block looks as follows.

$$\begin{aligned}
 &\langle \text{embedded-statement} \rangle \rightarrow \dots \mid \langle \text{waitone-statement} \rangle \mid \dots \\
 &\langle \text{waitone-statement} \rangle \rightarrow \text{waitone} [(\langle \text{waitone-timeout-expression} \rangle)] \{ \\
 &\quad \langle \text{future-expression} \rangle \} \\
 &\langle \text{waitone-timeout-expression} \rangle \rightarrow \langle \text{integer-expression} \rangle
 \end{aligned}$$

A waitone block can optionally take an integer expression enclosed in parentheses for specifying a wait timeout. If this expression is omitted, an infinite timeout is used. A possible implementation of the waitone block in pseudo-code is depicted in listing 4.5. This listing assumes that there are two operations, `wait` and `success`, for waiting on a future and checking the state of a future, respectively. A waitone block can operate on both non-generic and generic futures. The return value of a waitone block is an object of type `future` or `future<T>`, depending on the passed future expression.

```

1 waitone(f, timeout):
2     if f <> null
3         if wait(f, timeout) and success(f)
4             return f
5
6     return null

```

Listing 4.5: `waitone` Behavior

waitall The `waitall` keyword identifies a so called *waitall block*. As its name implies, a waitall block expects a non-empty list of future expressions and waits on the combined all group of the evaluated futures. The return value of the waitall block thereby depends on the fact whether this future wait operation was successful or not. In the former case, the return value is the evaluated list of futures of the passed future

expressions and in the latter case, the return value is `null`. Syntactically, a waitall block looks as follows.

```

⟨embedded-statement⟩ → ... | ⟨waitall-statement⟩ | ...
⟨waitall-statement⟩ → waitall [ ( ⟨waitall-timeout-expression⟩ ) ] { ⟨future-
expression-list⟩ }
⟨waitall-timeout-expression⟩ → ⟨integer-expression⟩

```

A waitall block can operate on both non-generic and generic futures. The passed future expression list must evaluate to future expressions of the same constructed generic type [18, p. 387] or to a list of non-generic value-less future expressions. A mixed list between non-generic and generic futures or a list with different constructed generic future types is not allowed. The return value of a waitall block is an object which implements the `IList<future>` or `IList<future<T>>` interface, depending on the passed future expressions.

Just like a waitone block, a waitall block can optionally take an integer expression enclosed in parentheses for specifying a wait timeout. If this expression is omitted, an infinite timeout is used. A possible implementation of the waitall block in pseudo-code is depicted in listing 4.6. Apparently, the waitall operation is a $\Theta(n)$ time and $\Theta(n)$ space operation where n denotes the count of futures expressions.

```

1 waitall(futures, timeout):
2     f <= all(futures)
3
4     if f <> null
5         if wait(f, timeout) and success(f)
6             return futures
7
8     return null

```

Listing 4.6: `waitall` Behavior

waitany The `waitany` keyword identifies a so called *waitany block*. As its name implies, a waitany block expects a non-empty list of future expressions and waits on the combined any group of the evaluated futures. The return value of the waitany block thereby depends on the fact whether this future wait operation was successful or not. In the former case, the return value is one successful future of the evaluated list of futures and in the latter case, the return value is `null`. Syntactically, a waitany block looks as follows.

```

⟨embedded-statement⟩ → ... | ⟨waitany-statement⟩ | ...
⟨waitany-statement⟩ → waitany [ ( ⟨waitany-timeout-expression⟩ ) ] {
⟨future-expression-list⟩ }
⟨waitany-timeout-expression⟩ → ⟨integer-expression⟩

```

A waitany block can operate on both non-generic and generic futures. The passed future expression list must evaluate to future expressions of the same constructed

generic type [18, p. 387] or to a list of non-generic value-less future expressions. A mixed list between non-generic and generic futures or a list with different constructed generic future types is not allowed. The return value of a waitany block is an object of type `future` or `future<T>`, depending on the passed future expressions.

Just like a waitone block, a waitany block can optionally take an integer expression enclosed in parentheses for specifying a wait timeout. If this expression is omitted, an infinite timeout is used. A possible implementation of the waitany block in pseudo-code is depicted in listing 4.7. Apparently, the waitany operation is a $\Theta(n)$ time and $\Theta(n)$ space operation where n denotes the count of futures expressions. If the passed future expression list contains only a single future expression, a waitany block behaves exactly like and can be replaced by the compiler for optimization reasons with a slightly more efficient waitone block.

```

1  waitany(futures, timeout):
2      f <= any(futures)
3
4      if f = null
5          return null
6
7      if wait(f, timeout) and success(f)
8          foreach g in futures
9              if success(g) return g
10
11     return null

```

Listing 4.7: `waitany` Behavior

4.1.3 Active Blocks

Several extensions can be added to the C[#] language and compiler to declare active blocks in a more natural and simpler way. The proposed language extension maps closely to the syntax of anonymous methods in C[#], i.e. the statements of an active block are enclosed in curly braces and identified by the `active` keyword. The following listing reuses an example from chapter 3.3.3 and replaces the explicit usage of the `Tasks` class with the corresponding C[#] keywords and language extensions.

```

1  future f = active { Array.Sort(s); }
2
3  /* Do some other work .. */
4  ...
5
6  f.Wait(); /* When the array is needed again */

```

Listing 4.8: Active Block Example

active The second purpose of the `active` keyword (compare section 4.1.1) is to act as an identifier for active blocks. The statements inside such an active block

are executed asynchronously as a whole and can optionally return a result value. Whether an active block computes a result value or not is determined by checking if it contains one or more return statements which return a result value. As with methods of active classes, result values of active blocks are modeled with future objects. Active blocks which are found not to compute a result also return a value-less non-generic future. Active blocks can be nested and embedded into other blocks. The syntactical changes for active blocks are as follows.

$$\begin{aligned} \langle \text{embedded-statement} \rangle &\rightarrow \dots \mid \langle \text{active-block} \rangle \mid \dots \\ \langle \text{active-block} \rangle &\rightarrow \text{active } \{ [\langle \text{statement-list} \rangle] \} \end{aligned}$$

The return value management of an active block is handled transparently in that necessary future objects are created, returned and filled automatically. The code inside an active block can be written without being aware of the future concept. The following points summarize a possible active blocks implementation.

1. An active block can basically be seen as an ordinary anonymous method with implicit asynchronous execution semantics. This means that the underlying implementation of an active block can be very similar to an anonymous method whose resulting delegate is executed implicitly and asynchronously.
2. Before scheduling the delegate for asynchronous execution, a future object is created and returned. A private reference to this future object is stored for later access. The created future object may be a non-generic or generic future, depending on the content of the active block as already mentioned.
3. After the active block has finished, the stored future object is filled. If the active block has failed, i.e. it has thrown an exception, the related exception object is stored in the future and its state is set to **failed**. Otherwise, its state is set to **succeeded** and, in the case of a generic future, the computed result of the active block is attached.

Since active blocks are modeled to return future objects, they can be combined with the previously introduced future related language extensions, namely `all`, `any`, `waitone`, `waitany` and `waitall` blocks. An example of how this can look like can be found in the following listing 4.9.

```

1  waitall {
2      active { Array.Sort(s); },
3      active { Array.Sort(t); }
4  }
```

Listing 4.9: Active Blocks and Futures

4.2 Benefits

Adding support for active objects, futures and active blocks to the programming language itself has several advantages over directly using the runtime library combined with the active object compiler approach. This section lists some of the benefits of a possible C[#] language extension.

First, it is simpler and less time-consuming to write active classes. The intermediate step of generating the public proxy interface code with the help of the active object compiler would be unnecessary. This is arguably the most important reason for a possible C[#] extension. Although it is theoretically possible to integrate the code generation step into development environments (as an IDE wizard or as part of the build process), a C[#] compiler extension would still be more convenient to use.

Furthermore, the active class and active block code is more readable. Inline `guard` blocks of active classes are easier to read and understand than dedicated guard methods that are annotated with the related attribute. Moreover, the syntax of active blocks let readers concentrate on the actual code inside and around these blocks and also abstracts away most of the details. This is especially true for developers who are not aware of the characteristics of the underlying Active Object model. Related to this is the fact that the implementation details of an active class can be hidden to a large extent. With the active object compiler or a manual implementation, the separation of an active object into the proxy and servant parts and the roles of the dispatcher and messages are still noticeable and visible. Not only that these details are of secondary importance in practice and only add unnecessary complexity, they also tie the active class code to the concrete implementation of the active object runtime library. A language extension on the other hand would allow for writing active object code which is portable and independent of the underlying implementation.

Last but not least, a compiler extension could do several additional things like analyzing the active class and active block code and warn about possible misuses and potential problems. While this could also be accomplished with a static code analyzer tool to some extent, integrating this functionality into the C[#] compiler itself seems to be the more natural and helpful way to do this. One example for this is the problem of public fields whose usage is illegal in an active class (compare section 4.1.1) according to my language extension proposal but are allowed in standard C[#].

None the less, the active object compiler has its uses as well. First of all, it is easier and faster to develop a simple code generator than to change a fully-fledged and complicated compiler. Although in the end it would be more convenient to have support for active objects, futures and active blocks directly in the programming language, the active object compiler is sufficient for testing purposes. Secondly, the active object compiler can be used to understand and study the Active Object pattern. This can come in handy when interested in the internal operation of active objects or the concrete implementation.

Chapter 5

Evaluation

This chapter is dedicated to evaluating the Active Object model. I begin with presenting it as a general high-level abstraction for traditional threading models and APIs and then proceed with giving several typical use-case scenarios for active objects, futures and active blocks. I then go on by discussing the performance implications of the Active Object model before introducing several advanced future improvements which are neither implemented in the example runtime library nor mentioned in my C[#] language extension proposal. I then conclude this chapter with a short section on the limitations and shortcomings of the Active Object model.

5.1 Abstraction

The Active Object model can be seen as a direct replacement or abstraction for a more explicit and more complicated threading model and API. In traditional multithreaded applications, threads are usually modeled with dedicated classes which inherit from a special thread class or implement a special thread callback method. These explicit low-level techniques are no longer necessary with the Active Object model since the entire thread management is handled transparently and is completely hidden when implementing an active class or using an active block. Not only that this results in less and easier to understand code, it can also help to prevent typical programming errors which can be ascribed to a more complicated threading model and API.

The difference between traditional threading models and the Active Object pattern is similar to comparing imperative programming with declarative programming. In the former case, programs tell the runtime environment exactly *how* to solve a certain task whereas in the latter case, a program merely specifies the *what*, i.e. the ultimate goal but not the concrete implementation. The *how* in our case is similar to using a threading API directly: declaring a class which implements a certain interface or inherits from a certain base class and makes use of the most basic synchronization primitives such as monitors or semaphores. Everything which is related to concurrent or multithreaded programming in this model is explicitly spelled out and there is

neither room for possible optimizations nor for using a different way to achieve the same goal. The Active Object model on the other hand is more like the what. You merely specify that a certain task can be carried out in parallel or that a certain object should run in its own thread of control but the concrete implementation is omitted. This is true even if you use the active object runtime library directly instead of the more abstract language extension. Also, instead of relying on low-level synchronization mechanisms such as locks and condition variables, there are now the abstractions of inherently thread-safe objects, futures and guards which are not only simpler to understand but also easier and more reliable to use.

Let's illustrate these statements with a short example (compare listing 5.1). For this purpose, I reuse the abstract producer-consumer example from Sutter [24]. The producer *produces* items in a loop and then passes these items to a consumer which *consumes* them.

```
1 active class Consumer {
2     public void Process(Message msg) {
3         ...
4     }
5 }
6
7 active class Producer {
8     ...
9     public void Produce() {
10        for (int i = 0; i < ...; i++) {
11            Message msg = ...;
12            consumer.Process(msg); /* Non-blocking */
13        }
14    }
15 }
```

Listing 5.1: Producer Consumer Example

Both the producer and consumer are active objects and run asynchronously with respect to each other. Produced items are automatically queued up if the producer generates more items than the consumer can handle. The necessary thread and synchronization management thereby is handled implicitly by the active object runtime library. Specifying that the tasks of the producer and consumer can be carried out in parallel is straightforward and is done simply by declaring the related classes as active classes. Contrast this with a more explicit threading model where both classes would need to follow the threading API specific implementation guidelines to manually start and stop threads as well as to manage and synchronize the intermediate queue of items by themselves.

5.2 Uses

After presenting the Active Object model as a general abstraction for a more advanced but also more complicated and verbose threading model and API, I use this

section to introduce several additional and more concrete scenarios. In addition to using the Active Object model to introduce concurrency into applications in order to maximize the overall processor throughput, there are other useful use-cases including modeling objects for asynchronous I/O or reducing latency and blocking in graphical user interface applications. In the following subsections, I introduce, demonstrate and explain the most common concrete scenarios.

5.2.1 Algorithms

Parallelizing algorithms is a prime example of how concurrency can help to maximize the processor throughput and to improve the overall runtime performance. Several examples can be found in the literature, some of which are discussed exemplary in this section. The Active Object model can help to express these parallel algorithms in a given programming language. The concept of active blocks to process certain tasks asynchronously and the synchronizations points in form of futures are well-suited to implement a variety of parallel algorithms.

Listing 5.2 demonstrates this with a simple parallel sort algorithm written in pseudo-code. It is essentially a parallel variant of the traditional merge sort algorithm [11]. It takes an array, divides it into several logical partitions and then sorts these partitions in parallel. After that, the sorted partitions are merged to form a single sorted array. The final merge operation thereby can theoretically be processed in parallel as well. Concurrency is achieved with active blocks and the necessary synchronization before the merge step is handled by waiting on the combined all future group of the active blocks.

```
1  sort(array):
2      partitions <= ... /* Divide array into partitions */
3
4      f <= null
5      foreach p in partitions
6          f <= and(f, active(seq_sort(p)))
7
8      waitone(f) /* Synchronization point */
9      merge(array, partitions)
```

Listing 5.2: Parallel sort

This procedure of dividing a task into smaller independent subtasks and then processing these subtasks in parallel is typical for a variety of parallel algorithms. For instance, a parallel search operation could look very similar to the sort example above. It would divide an array into smaller subarrays and then search these arrays in parallel. Unlike a parallel sort operation, a parallel search does not need to wait for all subtasks but can exit immediately once one of its subtasks has finished successfully.

Another typical parallelizable algorithm deals with loops. Consider a function which takes as arguments a list and another function and is responsible for applying this function to each element in the given list. Since each of these function calls is

independent from the others, they can be carried out in parallel. Such a function is well-known in several programming languages and environments and is usually called `map`. A possible parallel implementation of `map` with the Active Object model in pseudo-code is depicted in listing 5.3.

```

1 map(list, func):
2   if length(list) = 0
3     return [] /* Empty list */
4
5   f <= null
6   for i <= 0 to length(list) - 1 do
7     g <= active(a[i] <= func(list[i]))
8     f <= and(f, g)
9
10  waitone(f) /* Synchronization point */
11  return a

```

Listing 5.3: Parallel map

Depending on the passed function and its performance, it can make sense to divide the list into partitions rather than processing the list on a per-element basis in order to account for the additional runtime overhead caused by the thread management. If the passed function is fast enough, processing each item in parallel is usually slower in practice when compared to a more conservative partition-based approach. This is discussed in more detail in section 5.3.

Combined with another function usually called `reduce` or `fold` which takes a list, a binary function and an initial value and reduces the given list to a single result value, `map` allows for a broad range of applications. For instance, consider the example of computing the expected value or mean for a list of integers. The mean for such a list with n elements is commonly known as $\mu = \sum_i x_i p_i = \sum_i x_i P(X = x_i)$ where x_i stands for the value of the i th element in the list and $P(X = x_i)$ for the probability of this element. A possible parallel implementation of computing the mean for a given list in pseudo-code can be found in the following listing 5.4.

```

1 mean(list):
2   if length(list) = 0 then
3     return 0
4
5   a <= map(list, x => x * p(x))
6   return reduce(a, (x, y) => x + y, 0)

```

Listing 5.4: Parallel mean

5.2.2 Data Structures

The next use-case of the Active Object model I would like to discuss are data structures. Data structures and the related operations on these data structures are ideal candidates to be handled in parallel. Data structure operations are usually processor bound and also relatively costly in terms of processor time and can thus

benefit a lot from being processed asynchronously. The implementation thereby can range from a simple object-level concurrency model to a more fine-grained model on the operation-level. The latter case, i.e. introducing concurrency on the operation-level rather than on the object-level, overlaps to a certain degree with the parallel algorithm discussion from the previous section.

Recall from the introduction of chapter 2 that object-level concurrency is achieved with active classes whereas operation-level concurrency is implemented by means of active blocks.

For instance, consider the list example in listing 5.5. Since `List<T>` is declared as an active class, it provides object-level concurrency. Depending on the runtime behavior of the individual operations, one could choose to add additional operation-level concurrency to certain operations. For example, the `Remove` operation for removing an existing item from the list is usually a costly operation since it has to search the passed item in the internal list before it can remove it. But as explained in the previous section, this search procedure is well-suited for asynchronous execution and the `Remove` operation is therefore an ideal candidate for additional operation-level concurrency.

```
1  active class List<T> {
2      ...
3
4      public void Add(T item) {
5          ...
6      }
7
8      public void Remove(T item) {
9          ...
10     }
11 }
```

Listing 5.5: `List<T>` Class

A different example for an asynchronous data structure class is the `Buffer` class from the preceding chapters 3 and 4. It uses a strict object-level concurrency model, i.e. it is implemented as an active class but does not provide any additional operation-level concurrency. This makes sense in this case since the individual operations for enqueueing and dequeueing items are not really parallelizable and a more fine-grained model is therefore not easily applicable.

Strict object-level concurrency model is also the preferred model if an existing data structure is transformed into a parallel data structure by providing a wrapper type around the original class. If we have a strictly sequential data structure class, we can add a parallel version of the same class by implementing an active class which does nothing more than to call the sequential operations of the original data structure. Operation-level concurrency is not applicable here since the behavior of the operations of the original data structure cannot be changed in this scenario.

5.2.3 Asynchronous I/O

I/O is short for input/output and is the collective term for all things which are related to the input and output of a computer system. Examples for input operations include reading user input via an input device such as a keyboard or a mouse, reading data from a storage device or receiving data via the network. Typical output operations on the other hand are displaying information to the user and writing or sending data to a storage or network device. I/O related operations are usually very costly, not necessarily in terms of processor time but more in terms of waiting time since an application is normally blocked while performing an I/O operation. This in turn results in wasting processor time. One general way to solve this problem is to perform these operations asynchronously, i.e. in the background in a different thread of control and without blocking the application.

Listing 5.6 shows an example of this with an asynchronous I/O class which uses the object-level concurrency model. The class is called `Log` and is responsible for writing a log message to the console. By declaring `Log` as an active class, its write operation is not only performed asynchronously but also thread-safely. Thread-safety is an equally important property in this case since this guarantees that two write operations do not interfere with each other, even if the underlying implementation of writing a message to the console is not thread-safe by itself. Performing the write operation asynchronously on the other hand ensures that the calling thread is not blocked and can do other things in the meantime. Issued log messages thereby are automatically queued up if necessary. Recall from section 2.1.1 that the messages are guaranteed to be dequeued in chronological order which is a critical requirement in the case of a logging class.

```
1  static active class Log {
2      public static void Write(string message) {
3          System.Console.WriteLine(message);
4      }
5  }
```

Listing 5.6: Asynchronous Log

5.2.4 User Feedback

The next scenario uses the Active Object model in a slightly different way. This time, I demonstrate a typical technique to reduce the latency and improve the responsiveness in a user-controlled environment such as a graphical user interface application. This differs from previous use-cases where concurrency was mostly introduced in order to maximize the overall processor throughput and improve the runtime performance.

See listing 5.7 for an abstract example in which a long running task encapsulated in an active block is scheduled for background execution and the returned future is used for polling. The poll loop thereby is responsible for giving feedback about the background task to the user. This task could be any task which takes some time to

complete and blocks the application while executing. A practical example for this use-case scenario could be downloading a file in the background while simultaneously updating a progress bar.

```
1 future f = active { /* Some long running task */ }
2
3 while (f.IsPending) {
4     /* Do some other work like updating the UI */
5     ...
6     f.Wait(100); /* Wait 100 milliseconds */
7 }
8
9 if (f.HasSucceeded) {
10     ...
11 }
```

Listing 5.7: Running a Long Task

5.3 Performance

Something that I have not discussed in this thesis in detail so far is the expected performance of applications which use the Active Object model. Since this is obviously an important point in the whole Active Object discussion, I use this section to analyze the impacts of active objects, futures and active blocks on the runtime performance of applications.

When seen from a high-level perspective, the Active Object model is meant to introduce concurrency into applications with the intention of improving their runtime performance and maximizing the overall processor throughput. In contrast to strictly sequential applications which are limited to using a single core or processor only, concurrent applications may scale to use all available cores at once. For instance, consider two different applications s and c which for the same input always emit the same output. The first application s is a strictly sequential application which at any given time has a single thread only. The second application c on the other hand is a concurrent application which utilizes all available cores. If we let run both applications with the same input on two identical machines with n cores each, this means that c *might* finish earlier by the same factor n in the best case.

One thing that is important to note when talking about performance considerations is that the Active Object model as presented in this thesis is merely meant to be an abstraction for expressing concurrent programming in a simpler way than traditional threading models and APIs are capable of. It does not, however, provide any performance related guarantees or an inherently scalable system. This means that it is still the responsibility of the application developer to use concurrency in a way that makes sense performance-wise. Otherwise, the non-negligible overhead of concurrency and multithreading can result in a runtime performance that is worse than the performance of a strictly sequential application. This is especially true for

machines with very few cores or processors. To use the Active Object model in an efficient way, it is thus important to consider the performance implications of using active objects and active blocks.

As explained previously in chapter 2, each active object and each active block conceptually runs in its own thread of control. Recall that it is not necessarily specified that two subsequent operations of an active object run in the same thread. It is merely guaranteed that these two operations are processed in a thread context different from that of a client. While this allows for several useful optimizations in practice by the runtime library such as reducing the thread creation and shutdown overhead (compare the thread pool solution in section 3.1.1), it does not change the fact that k simultaneously processed active object and active block operations result in having k active threads. Even for a reasonable large k , this in turn may result in a less than optimal runtime behavior and performance caused by the increased context switching and thread management overhead.

Aside from this general performance and scalability fact, another important point to consider is the overhead of the Active Object model itself. Because of the internal message-based behavior and component-based structure, using active objects involves creating several additional and intermediate objects behind the scenes which may have an impact on the runtime performance. To be more precise, depending on the lifetime of an active object and the average time spent in individual active object operations, the induced overhead can influence the runtime performance and processor throughput in a negative way. For instance, consider a read-only property or getter method of an active object which simply returns a private field to a client. The overhead of storing a client request in the dispatcher queue and creating the related intermediate future and message objects would exceed by far the actual time spent in this property or method. This should be taken into account when designing active classes and writing active blocks.

5.4 Future Work

Although the Active Object abstraction as presented in this thesis is already an improvement over traditional threading models and APIs as explained in section 5.1, there are a few features and characteristics missing to make it applicable to real-world applications.

The performance and scalability related issues explained in the previous section can make it difficult under certain circumstances to use the Active Object model in an efficient way. Moreover, some features and properties which give a bit more control about the runtime behavior of active object operations and active blocks may be necessary depending on the intended usage. In this section, I will summarize several enhancements and advanced changes to the standard Active Object model which make it an inherently scalable, feature-rich and more efficient concurrency abstraction.

5.4.1 Active Object Pools

Active object pools [12, p. 10] are an advanced concurrency concept and are built on top of active objects. Externally, an active object pool appears and behaves exactly like a traditional active object. It provides a standard strongly-typed proxy interface with normal methods and properties which can take parameters and return future objects. Internally, however, an active object pool acts more like a *set* of active objects. That means, instead of processing at most one client request at any given time, an active object pool can process multiple client requests simultaneously.

Implementation-wise, this is achieved by maintaining a $1 : n$ relationship between the public proxy interface and an internal collection of active objects. Once a method of the proxy is invoked by a client, the pool chooses a possibly idling active object and forwards the request to the related dispatcher. Since it is not necessarily specified which internal active object is chosen, an active object pool is mostly intended for processing stateless operations. The amount of internal active objects in an active object pool may grow or shrink dynamically depending on certain criteria such as the current processor load or amount of active threads. Listing 5.8 shows a possible example for an active object pool.

```
1 active pool Workers {
2     ...
3     public void Process(WorkItem item) {
4         ...
5     }
6 }
```

Listing 5.8: Active Object Pool

Active object pools can be seen as a high-level abstraction for thread pools known from traditional threading models and APIs, hence the name pool in the first place.

5.4.2 Scalability

The most important performance-related enhancement for the Active Object model concerns its scalability. As explained in the preceding section 5.3, the standard Active Object model does not provide any guarantees about its performance or scalability. This is mostly due to the fact that active object operations and active blocks are always processed asynchronously in a private thread of control. This allows for a simple theoretical model and implementation but does unfortunately not scale very well in practice.

I therefore propose two additional concurrency primitives, *activable objects* and *activable blocks*. As their name imply, activable objects are objects that may behave like an active *or* passive object. Similarly, activable blocks may behave like an active *or* traditional passive code block. Thus, activable objects and activable blocks have nearly the same properties as active objects and active blocks with the minor but important difference that their operations are no longer *guaranteed* to but merely

may be executed asynchronously. The decision whether an activable object or activable block is treated as active or passive entity is made on a per-request basis. This means that a client request to an activable object or an activable block may either be executed asynchronously or sequentially in the thread context of the client. It is important to note that, despite this change in the execution semantics, activable objects are still fully thread-safe. Client requests are still enqueued and processed sequentially with respect to each other even when not processed asynchronously.

Weakening the strict asynchronous property of traditional active objects and active blocks has the benefit that the runtime can now dynamically and automatically decide whether processing a certain request asynchronously or not make sense from a performance perspective. Similar to active object pools, this decision may be based on several criteria such as the current processor load or the current amount of active threads. Ideally, this then results in a nearly optimal runtime behavior and performance.

It is important to note that activable objects and activable blocks do not replace the concepts of traditional active objects and active blocks but merely add two additional abstractions to the Active Object model. Since there are certain scenarios where strict asynchronous execution is a requirement such as the user feedback example from section 5.2.4 or when working with guards or condition variables, all four abstractions are necessary. Syntactically, the differentiation between activable classes and activable blocks on the one hand and traditional active classes and active blocks on the other hand may be realized with a new `activable` keyword which acts both as a class modifier for activable classes and as an identifier for activable blocks.

5.4.3 Efficiency

Aside from the general scalability-related change introduced in the preceding section, there are two additional important optimization techniques which positively influence the runtime performance of the Active Object model.

First, instead of using the pre-defined components of the active object runtime library, a language or active object compiler could choose to integrate the dispatcher functionality directly into the generated proxy of an active class (also compare section 2.1.1). In addition to reducing the amount of created objects at runtime, an integrated custom model could omit support for priorities and guards as long as the related active class does not use these features in order to improve the runtime performance even further.

The second optimization technique is to always execute certain active object operations or active blocks sequentially. As outlined in section 5.3, the overhead of the Active Object model becomes the more noticeable, the less time is spent in an active object operation or active block. The runtime can, either with the help of application developers or with runtime heuristics, optimize away these situations to some degree by processing certain requests sequentially. The only thing the runtime must take care of in this case is that the thread-safety property of active objects is still preserved.

5.4.4 Dynamic Priorities

Dynamic priorities are an advanced feature of the dispatcher queue and are intended for automatically adjusting the priorities of enqueued requests. They are used to *promote* certain active object operations to get a higher priority than originally specified in case they have already been waiting for a long(er) time in the dispatcher queue. Similar to operating system processes and schedulers, the dispatcher can increase the priority of certain operations in order to prevent *request starvation*, i.e. a client request is never dequeued and processed because of other constantly enqueued higher-priority requests.

5.5 Limitations

Active objects are applicable to a wide range of problems but also have their limitations. One limitation deals with circular object references. Circular object references (direct or indirect) can introduce deadlock situations under certain circumstances and are best avoided with active objects. For example, consider the following scenario with two active objects p and q and two futures f and g .

1. p calls a method of q and waits on the returned future f . This method request is enqueued and eventually executed in the thread context of q .
2. q in turn holds a circular reference to p and invokes a method of p in the original method request of p . q also waits on a returned future g .
3. Since p is blocked while waiting on the future f of q , the method request of q can never be processed. Thus, both objects are blocked and experience a deadlock situation.

Figure 5.1 shows the corresponding message sequence chart for this deadlock scenario. It is clearly visible that both objects are deadlocked after the second active object q has entered the wait state.

Deadlocks can be prevented by not using circular references or, if this is not possible or desired, by ensuring that the described situation cannot occur by avoiding the possibility of having both active objects wait for each other. No deadlock can occur if at most one active object waits for the other since this wait operation eventually ends assuming that the other active object is not deadlocked otherwise.

Another limitation of active objects deals with the fact that the servant is usually unaware of the active object context it is used in and does not know anything about its proxy wrapper. While this is usually perceived as a benefit of the Active Object model as outlined in section 2.1.1, it can cause problems if a servant needs to pass a reference of itself to another object. In this case, the servant would break the active object abstraction and encapsulation thus partially losing properties like asynchronous method execution and thread-safety. This can be worked

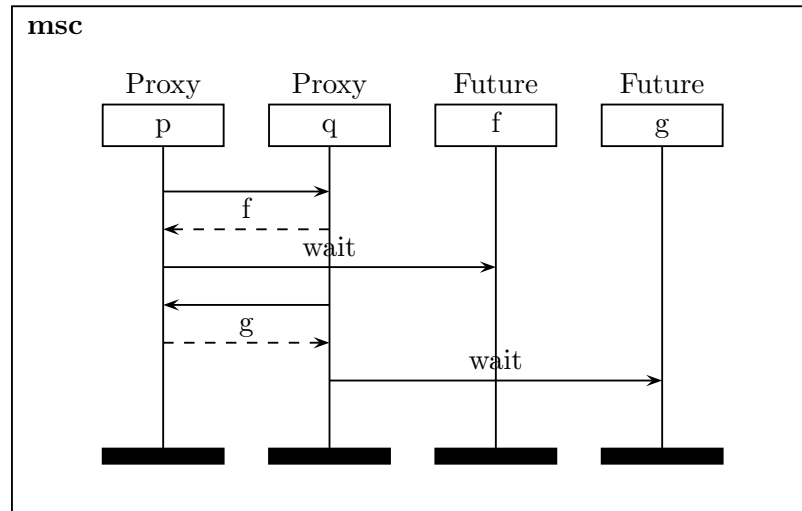


Figure 5.1: Active Object Deadlock

against by adding support for active objects directly to programming languages and environments but is difficult or impossible to achieve with a runtime library.

Slightly related to this is the inheritance and interface compatibility problem. As is known by now, return values and out parameters of active methods are modeled with future objects. This results in changing the original class signature of a servant and leads to incompatible interfaces. While this is usually not a problem, it may be a reason for not being able to let the proxy implement a certain interface or inherit from a particular abstract base class. Even in those cases when deriving from an abstract class is possible language-wise, methods declared in the base class do not necessarily follow the same active objects semantics as the deriving class.

Chapter 6

Conclusion

In this thesis, I presented the Active Object model as one possible way to simplify today's concurrent programming. I started in chapter 2 with the theoretical aspects of the model and explained the underlying structure, behavior and interfaces in detail. After that, in chapter 3, I introduced the active object runtime library and the active object compiler, my attempts to implement the Active Object model in .NET and C#. In the subsequent chapter 4, I presented a proposal for integrating the Active Object model directly into C# itself as opposed to using a third-party library as well as an additional code generator tool. In chapter 5, I then evaluated the Active Object model in general and my implementation in particular. I gave several typical use-case scenarios and pointed out the advantages as well as disadvantages of the Active Object model and my implementation.

Summarizing this whole thesis, the Active Object model allows for introducing concurrency into applications at a higher abstraction level than traditional threading APIs and models. By adding several concurrency abstractions on top of existing low-level concepts, the Active Object model largely removes the need for using a more verbose, error-prone and complicated threading model and thus simplifies concurrent programming. One major benefit of the Active Object model is indeed the fact that it can introduce concurrency even in otherwise strictly sequential situations. It is actually quite straightforward to take a big task, split it up into smaller chunks and then carry these chunks out in parallel (such as parallelizing a for-loop or a recursive algorithm) but it is a different challenge to create an inherently concurrent system. The Active Object pattern is well-suited to achieve this by being able to model concurrency not only on the operation-level but also on the object-level. Due to the mentioned limitations and problems, the Active Object model might not be the universal solution to all concurrent problems, but it definitely has the potential to become one of several new promising tools for concurrent programming.

It will be interesting to see how concurrent programming will be integrated into the main programming languages in the upcoming years. I guess that the Active Object model or similar high-level abstractions will eventually find its way to mainstream programming languages. While working on this thesis, for instance, Microsoft has

released a first preview version of its upcoming Parallel FX Library [13] for the .NET framework. It shares several concepts with the Active Object model such as futures and, to some extent, also the concept of active blocks (in the form of tasks). It would be interesting to see a version of the Active Object model implemented on top of this new framework.

Bibliography

- [1] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*, section 7.3. Addison-Wesley, 2000.
- [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 6.5. The MIT Press, second edition, September 2001.
- [3] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, ed., *Programming Languages*, pp. 43–112. Academic Press, 1968.
- [4] Dino Esposito. Design and develop seamless distributed applications for the common language runtime. *MSDN Magazine*, October 2002.
- [5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.
- [6] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Prentice Hall, third edition, 2005.
- [7] William Grosso. *Java RMI*. O'Reilly, 2001.
- [8] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, volume 17, no. 10, pp. 549–557, October 1974.
- [9] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall International, second edition, 1988.
- [10] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*, section 2.3. Addison-Wesley Professional, third edition, 1997.
- [11] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*, section 5.2.4. Addison-Wesley Professional, second edition, 1998.
- [12] R. Greg Lavender and Douglas C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Proc. Pattern Languages of Programs*, 1995.

-
- [13] Daan Leijen and Judd Hall. Optimize managed code for multi-core machines. *MSDN Magazine*, October 2007.
 - [14] Mike Repass. Reflections on reflection. *MSDN Magazine*, June 2007.
 - [15] Jeffrey Richter. Microsoft .net framework delivers the platform for an integrated, service-oriented web. *MSDN Magazine*, September 2000.
 - [16] Jeffrey Richter. Type fundamentals. *MSDN Magazine*, December 2000.
 - [17] Jr. Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, volume 7, no. 4, pp. 501–538, 1985.
 - [18] Standard ECMA-334. *C[#] Language Specification*, June 2006.
 - [19] Standard ECMA-335. *Common Language Infrastructure*, June 2006.
 - [20] W. Richard Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI*. Prentice Hall PTR, 1997.
 - [21] W. Richard Stevens. *UNIX Network Programming: Interprocess Communications*. Prentice Hall PTR, 1999.
 - [22] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, special third edition, 2000.
 - [23] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs' Journal*, volume 30, no. 3, 2005.
 - [24] Herb Sutter. The concur project: Some experimental concurrency abstractions for imperative languages, 2006.
 - [25] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, volume 3, no. 7, pp. 54–62, 2005.