University of Oslo
Department of Informatics

# Liberating Coroutines: Combining Sequential and Parallel Execution

**Master thesis**

Steingrim Dovland

**31st January 2006**

## Abstract

Concurrent programming using threads is considered a hard and error-prone task. Coroutines are conceptually simpler, they are easier to program with due to their sequential nature. Flexible coroutines as presented by Belsnes and Østvold liberate classical coroutines from their quasi-parallel world and combine them with threads. This allows the programmer to factor programs into sequential and parallel tasks, leading to simpler programs.

This thesis presents an extension to the formal semantics for flexible coroutines. A detailed breakdown of the scheduling strategies and parameter passing is presented in the same formal framework. Some words are given on patterns that emerge when programming with flexible coroutines and these patterns are defined in the formal framework.

We present a clean implementation of flexible coroutines in Java, based on standard threads and semaphores. Challenges encountered, such as representing coroutines in Java and invoking methods across threads are discussed. This framework is used in examples that employ flexible coroutines in different ways; the classical synchronization problem of readers and writers, the Santa Claus problem and binary and general semaphores.

# Contents

**1 Introduction** **1**

1.1 Objectives . . . . . 1

1.2 Organization of thesis . . . . . 2

**2 Coroutines** **5**

2.1 Classification of coroutines . . . . . 5

2.1.1 Control transfer mechanisms . . . . . 6

2.1.2 First-class object or a language construct . . . . . 8

2.1.3 Stackfullness . . . . . 9

2.1.4 Full coroutines . . . . . 9

2.2 Simula coroutines . . . . . 9

2.3 Similar constructs in other languages . . . . . 11

2.3.1 Lua coroutines . . . . . 11

2.3.2 Python generators . . . . . 12

2.3.3 Scheme continuations . . . . . 12

**3 Java thread model** **15**

3.1 Threads in Java . . . . . 15

3.2 Java object monitors . . . . . 17

3.3 Java concurrency utilities . . . . . 19

3.3.1 Locks and condition variables . . . . . 19

3.3.2 Semaphores . . . . . 20

**4 Semantics of flexible coroutines** **23**

4.1 Notation . . . . . 23

4.2 Basic operations . . . . . 25
4.2.1 Coroutine-coroutine interaction . . . . . 26
4.2.2 Thread-coroutine interaction (*idle* coroutines) . . . . . 28
4.2.3 An invariant for active coroutines . . . . . 29
4.3 Scheduling . . . . . 30
4.3.1 The scheduling rule . . . . . 30
4.3.2 Transitions that lead to scheduling . . . . . 31
4.4 A local process set . . . . . 31
4.4.1 A meta-function for chasing the callee chain . . . . . 32
4.4.2 Thread-coroutine interaction (*waiting* and *active* coroutines) 33
4.4.3 The attachment rules . . . . . 35
4.5 Parameters and return values . . . . . 36
4.5.1 A parameter and return value substitution meta-function . 37
4.5.2 Parameters . . . . . 37
4.5.3 Return values . . . . . 39
4.6 Flexible coroutine patterns . . . . . 39
4.6.1 Spinning coroutine . . . . . 40
4.6.2 The attached-only coroutine . . . . . 40

**5 A flexible coroutine system in Java 43**
5.1 Overview . . . . . 45
5.1.1 A simple example . . . . . 46
5.1.2 Implementation overview . . . . . 47
5.2 Coroutine system . . . . . 49
5.2.1 System lock . . . . . 50
5.2.2 Scheduler . . . . . 50
5.3 Coroutine operations . . . . . 52
5.3.1 The Operation class . . . . . 53
5.3.2 Passivate . . . . . 55
5.3.3 Yield . . . . . 57
5.3.4 Kick . . . . . 58
5.3.5 Call . . . . . 59

5.3.6 Detach . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
5.4 Coroutine instances . . . . . . . . . . . . . . . . . . . . . . . . . 65
5.4.1 The coroutine runner thread . . . . . . . . . . . . . . . . . 65
5.4.2 Suspending and resuming coroutines . . . . . . . . . . . . 66
5.5 Exception handling . . . . . . . . . . . . . . . . . . . . . . . . . 68
5.6 Flexible coroutine patterns . . . . . . . . . . . . . . . . . . . . . 70
5.6.1 Spinning coroutine . . . . . . . . . . . . . . . . . . . . . . 70
5.6.2 The attached-only coroutine . . . . . . . . . . . . . . . . . 71
5.7 Status . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72

**6 Using flexible coroutines 73**
6.1 Readers and writers . . . . . . . . . . . . . . . . . . . . . . . . . 73
6.1.1 Prefer writer access . . . . . . . . . . . . . . . . . . . . . . 76
6.1.2 Prefer reader access . . . . . . . . . . . . . . . . . . . . . 77
6.2 The Santa Claus problem . . . . . . . . . . . . . . . . . . . . . . 79
6.2.1 An auxiliary class . . . . . . . . . . . . . . . . . . . . . . . 79
6.2.2 A solution . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
6.2.3 An alternative solution . . . . . . . . . . . . . . . . . . . . 82
6.2.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
6.3 Semaphores . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
6.3.1 Binary semaphore . . . . . . . . . . . . . . . . . . . . . . . 84
6.3.2 General semaphore . . . . . . . . . . . . . . . . . . . . . . 85

**7 Related work 89**
7.1 Sequential Object Monitors . . . . . . . . . . . . . . . . . . . . . 89
7.1.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
7.1.2 Implementation . . . . . . . . . . . . . . . . . . . . . . . . 90
7.1.3 Compared to flexible coroutines . . . . . . . . . . . . . . . 91
7.2 Polyphonic C♯ . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
7.2.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . 91

**8 Conclusion 93**
8.1 Contribution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93

8.2 Critique . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
8.3 Future work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95

**A Code listings 97**

A.1 Flexible coroutines in Java . . . . . . . . . . . . . . . . . . . . . . . 97
A.1.1 Coroutine.java . . . . . . . . . . . . . . . . . . . . . . . . . 97
A.1.2 SpinningCoroutine.java . . . . . . . . . . . . . . . . . . . . 106
A.1.3 AttachedOnlyCoroutine.java . . . . . . . . . . . . . . . . . . 106
A.2 Readers and writers . . . . . . . . . . . . . . . . . . . . . . . . . . 106
A.2.1 Reader and writer threads . . . . . . . . . . . . . . . . . . . 106
A.2.2 RWFIFO.java.tex . . . . . . . . . . . . . . . . . . . . . . . . 108
A.2.3 RWPreference.java . . . . . . . . . . . . . . . . . . . . . . . 109
A.3 The Santa Claus problem . . . . . . . . . . . . . . . . . . . . . . . 110
A.3.1 Rendezvous.java . . . . . . . . . . . . . . . . . . . . . . . . 110
A.3.2 NWay.java . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
A.3.3 SantaClaus.java . . . . . . . . . . . . . . . . . . . . . . . . . 111
A.4 Semaphores . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
A.4.1 BinarySemaphore.java . . . . . . . . . . . . . . . . . . . . . 114
A.4.2 GeneralSemaphore.java . . . . . . . . . . . . . . . . . . . . 115

# List of Figures

2.1 Symmetric control transfer mechanism. . . . . . . . . . . . . . . . . 7
2.2 Asymmetric control transfer mechanism. . . . . . . . . . . . . . . . 8
2.3 A dynamic hierarchy of coroutines. . . . . . . . . . . . . . . . . . . 10
2.4 The Simula program corresponding to figure 2.3. . . . . . . . . . . 11
2.5 Generator in Python . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.6 Closures in Scheme . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.7 Closures in Scheme . . . . . . . . . . . . . . . . . . . . . . . . . 13

3.1 The life cycle of a Java thread. . . . . . . . . . . . . . . . . . . . . 16
3.2 Java object monitor. . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.3 A simple Java class that acts like a monitor . . . . . . . . . . . . . 18
3.4 Bounded buffer using a lock and two condition variables. . . . . . 22

4.1 Formal notation summary. . . . . . . . . . . . . . . . . . . . . . . 26
4.2 State diagram for a flexible coroutine $n$. . . . . . . . . . . . . . . 27
4.3 State diagram for a thread $t$. . . . . . . . . . . . . . . . . . . . . . 29
4.4 Flexible coroutine and thread-interaction semantics. . . . . . . . 29
4.5 A flexible coroutine system invariant. . . . . . . . . . . . . . . . . 30
4.6 The scheduling rule. . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.7 Callee chain meta-function. . . . . . . . . . . . . . . . . . . . . . 33
4.8 Thread interaction on *waiting* and *active* coroutines. . . . . . . . 35
4.9 The attachment rules. . . . . . . . . . . . . . . . . . . . . . . . . 36
4.10 Parameter and return value substitution meta-function. . . . . . 37
4.11 Parameter passing and return values. . . . . . . . . . . . . . . . . 39

4.12 Semantics for special coroutines. . . . . . . . . . . . . . . . . . . . 41

5.1 Example usage of the Coroutine class. . . . . . . . . . . . . . . . . . 47
5.2 Implementation of the flexible coroutine system. . . . . . . . . . . 50
5.3 Two threads competing for the system lock. . . . . . . . . . . . . 51
5.4 Forced and non-forced scheduling of coroutines. . . . . . . . . . . 52
5.5 Implementation of schedule() method. . . . . . . . . . . . . . . . 53
5.6 Encapsulation of caller and parameters in the local queue of a coroutine. 54
5.7 Passivating $m$ awakens waiting coroutine $n$. . . . . . . . . . . . . 56
5.8 System queue and local queue relationship. . . . . . . . . . . . . 58
5.9 Implementation of call operation part 1. . . . . . . . . . . . . . . 61
5.10 Implementation of call operation part 2. . . . . . . . . . . . . . . 63
5.11 Implementation of detach operation. . . . . . . . . . . . . . . . . 64
5.12 The enter() method. . . . . . . . . . . . . . . . . . . . . . . . . 67
5.13 The leave() method. . . . . . . . . . . . . . . . . . . . . . . . . 68
5.14 The SpinningCoroutine pattern. . . . . . . . . . . . . . . . . . . . 71
5.15 The AttachedOnlyCoroutine pattern. . . . . . . . . . . . . . . . . 71

6.1 Readers and writers. . . . . . . . . . . . . . . . . . . . . . . . . 75
6.2 The local queue of the request coroutine. . . . . . . . . . . . . . . 77
6.3 Readers and writers with writers preference. . . . . . . . . . . . . 78
6.4 A binary rendezvous class. . . . . . . . . . . . . . . . . . . . . . 80
6.5 A multi-way rendezvous class. . . . . . . . . . . . . . . . . . . . 80
6.6 A solution to the Santa Claus problem. . . . . . . . . . . . . . . . 81
6.7 The Reindeer class. . . . . . . . . . . . . . . . . . . . . . . . . . 82
6.8 Synchronizing on Santa Claus and ringing his doorbell. . . . . . . 82
6.9 Santa Claus as a flexible coroutine. . . . . . . . . . . . . . . . . . 83
6.10 Binary semaphore. . . . . . . . . . . . . . . . . . . . . . . . . . 85
6.11 General semaphore. . . . . . . . . . . . . . . . . . . . . . . . . . 87

7.1 A sequential object monitor. . . . . . . . . . . . . . . . . . . . . 90

# Preface

This thesis is submitted to the Department of Informatics at the University of Oslo as a part of my Masters degree (M.Sc.).

## Acknowledgements

My supervisor, Bjarte Østvold, has been a great support and he has always been available with wise counsel. His knowledge and understanding in both parallel programming specifically and programming languages generally and his willingness to share his knowledge plus his patience with me has been at uttermost importance for me. I also wish to thank the Norwegian Computing Center for lending me office space and a computer.

I also wish to thank my former colleagues at the computer lab in Niels Henrik Abels hus, who have provided me with much assistance in typesetting this thesis in LaTeX. Special thanks to Thorkild Stray for finding flaws in my formal semantics and for his proofreading.

Finally, Karoline deserves special thanks for being patient and supporting when my spare time was non-existant!

# Chapter 1

# Introduction

Programming languages, their semantics and subtle differences have appealed to me since the beginning of my academic career. How problems can be solved in different languages, using different language constructs and different programming paradigms has been my main interest for many years. In this thesis I combine this interest with programming, which gives it a practical approach to a theoretical problem.

## 1.1 Objectives

In an unpublished article Belsnes and Østvold [2] presents a formal semantics for the coroutines in the Simula programming language and then introduces a new language construct, the *flexible coroutine*, coroutines cooperating with threads. This thesis extends the semantics given in that article with rules for scheduling, parameters and return values, and then completes it by presenting the implementation of flexible coroutines in Java.

Coroutines, as presented in the next chapter, execute sequentially. A set of coroutines that execute sequentially are sometimes referred to as executing in quasi-parallel. Threads however, are truly parallel, and many language constructs and programming techniques exist to aid the programmer in programming such systems since programming with threads is considered hard and error-prone. The subtle bugs introduced by the non-deterministic behaviour of parallel threads often lead to bugs that are hard to reproduce.

Flexible coroutines aim to *liberate* the coroutines from their quasi-parallel world by combining the inherently sequential execution of coroutine systems with the parallel execution of threads.

Introducing a new language construct often involves introducing a new programming paradigm. To complete this thesis an important part is to show what flexible coroutines are and how they can be used to solve problems involving multiple processes. The main objective is therefore threefold:

1. Extend the formal semantics for flexible coroutines to include rules for scheduling, parameter passing and return values.
2. Develop an implementation of a flexible coroutine system in Java.
3. Show how flexible coroutines can be used to solve a variety of problems involving synchronization issues and multiple processes.

The first goal is to extend the existing formal semantics so that it includes parameters and return values. This includes reasoning about the existing semantics so that we identify what we must add to reach this goal. This also includes defining rules for scheduling flexible coroutine systems. A motivation for studying such semantics are to model language where regular procedure calls are special cases of asymmetrical coroutine calls, e.g. a language where every object acts like a coroutine.

We aim to implement a reference framework for programming with flexible coroutines in Java. This framework should implement the semantics that we present, and also function as a supplement for the formal semantics.

The third goal is to use the framework developed to solve different kinds of problems by using flexible coroutines. By doing so we show that the semantics is useful and that the implementation presented can be used to solve real-life problems.

## 1.2 Organization of thesis

The following describes the structure of this thesis.

**Chapter 2 Coroutines** This chapter describes and defines the coroutine concept in an informal way. We will show a simple classification of what a coroutine is and then describe Simula coroutines in a detailed manner. Finally similar constructs in different programming languages are shown with simple examples and a discussion of how these relate to coroutines.

**Chapter 3 Threads in Java** In this chapter we take a look at the Java thread model and the different utilities provided to solve parallel programming problems. These utilities will be used in chapter 5 when implementing flexible coroutines in Java.

**Chapter 4 Semantics of flexible coroutines** This chapter presents the formal semantics for flexible coroutines. First we briefly discuss the basic operations. Then this is extended upon and more detailed semantics for parameter passing and return values are presented, as well as some special case coroutines.

**Chapter 5 Flexible coroutines in Java** This chapter presents a complete implementation of a flexible coroutine system in Java is given. We take a detailed look at the challenges faced when implementing such a system and how the choices made affect the overall design of the system and the usability and efficiency.

**Chapter 6 Using flexible coroutines** This chapter consists of three parts, each part shows an example of how to use flexible coroutines. First we present the Readers/Writers problem, then a more illustrative example, the Santa Claus problem, is discussed and solved using flexible coroutines. Finally we show how binary and general semaphores can be implemented using flexible coroutines.

**Chapter 7 Related work** This chapter presents two examples of related work in Java and C♯.

**Chapter 8 Conclusion** This chapter presents the conclusion and future work.

# Chapter 2

# Coroutines

Those who understand Simula best are not simply the programmers, but the users of the Simulation class.

Kristen Nygaard

In this chapter we take a close look at what defines a coroutine. The coroutine concept is quite old, found in languages such as Simula and Modula-2 and for a long period of time, coroutines have lived in a state of oblivion. However, in recent years, many modern programming languages have adopted concepts that provide similar features, some have even incorporated full-grown coroutines. This chapter starts of with a informal look at coroutines and tries to define what constitutes coroutine behaviour. Then we take a more detailed look at the coroutines provided by the Simula programming language, often regarded as the prototype implementation of coroutines. The chapter finishes of with presenting similar concepts in a variety of languages, namely coroutines in Lua, generators in Python and continuations in Scheme and we look at how these relate to the coroutine concept defined earlier in this chapter.

## 2.1 Classification of coroutines

The coroutine concept is usually [10] attributed to Conway, who described and implemented the construct in 1963 to simplify the cooperation between the lexical and syntactical analyzer in a one-pass COBOL compiler [8]. Since then coroutines have been implemented in many languages, most notably perhaps the coroutines implemented in Simula 67 [9] by Dahl and Nygaard. However despite being an old concept, coroutines as a control transfer component is much less known than other similar concepts.

When Knuth [28] explained coroutines he compared them with the less general, but more common program component, the subroutine. Whereas a regular subroutine has only one entrance point, at its “beginning”, the coroutine is always initiated *at the place following* where it last terminated. In his comparison he also shows a relation between coroutines and multiple-pass algorithms and filters.

In his doctoral thesis, Marlin [34] summarizes the fundamental characteristics of a coroutine as

- *“the values of data local to a coroutine persist between successive calls”*
- *“the execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage”.*

In short; a coroutine is a subroutine with state, a resumable subroutine. However, such a non-formal coroutine definition has led to many implementations with slightly different semantics. Ierusalimschy and de Moura designed and implemented coroutines in Lua (see section 2.3.1) and also discussed [10] the most notable differences between coroutine mechanisms. They identified three main issues:

- *“the control-transfer mechanism, which can provide symmetric or asymmetric coroutines”.*
- *“whether coroutines are provided as first-class objects [. . . ] or as constrained constructs”.*
- *“whether a coroutine is a stackful construct, i.e., whether it is able to suspend its execution from with nested calls”.*

As new programming languages evolved coroutines where adapted in several different manners. Most important is perhaps the first of these three points, regarding the control transfer mechanism. The remainder of this section describes these differences and their impact on the expressiveness of the language.

### 2.1.1 Control transfer mechanisms

The most notable classification of coroutines concerns the control transfer operations that are provided. A common approach is to distinguish between *symmetric* and *asymmetric* coroutines.

Symmetric coroutines provide only one control transfer mechanism, which allows the coroutines to explicitly pass control between themselves, figure 2.1 shows an example of how this might look like in an application with three coroutines called $main$, $m$ and $n$.

The operation provided for control transfer in this figure is called resume(, ) which is also the name used for the similar construct in Simula. When the main coroutine transfers control to $n$ its state is saved, meaning that the values of its variables will persist between the point where control left main and when

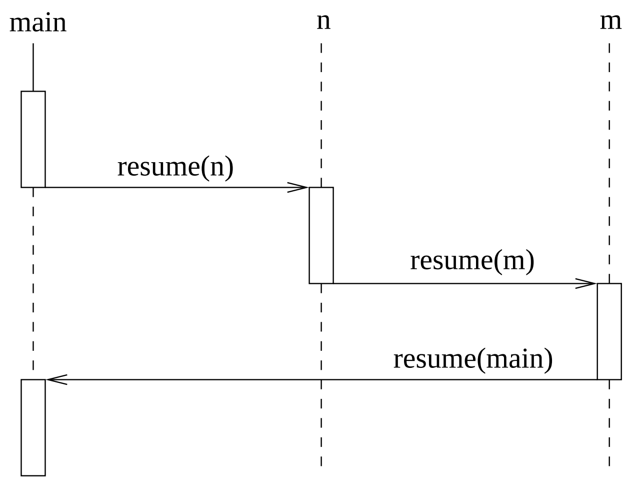

**Figure 2.1:** Symmetric control transfer mechanism.

it enters main again. Later on the coroutine named $m$ resumes main and at this point main carries on where it left of after having resumed $n$. In many ways this resembles the kind of programming that occurs with heavy use of the goto construct found in many programming languages. Although this is often regarded as a dangerous and harmful way of programming [14], it can also lead to structured and easy to read code when done correctly [27].

Another way of thinking of coroutines has led to the adoption of asymmetric coroutines, which provide two control transfer mechanisms. One for invoking a coroutine and one for suspending it, the latter which returns control to its invoker. Figure 2.2 shows an example of control flow in a program with three coroutines named $m$, $n$ and $k$ employing asymmetric control transfer mechanisms.

The operations provided for control transfer in this figure are named call$(a)$nd detachafter their counterpart-names in Simula. An important aspect of asymmetric coroutines is that a relationship is established between the caller and the callee. The effect is that when a coroutine performs detachcontrol is passed back to the coroutine that invoked the original coroutine. As we see in figure 2.2 this leads to the familiar caller/callee pattern that we see with regular functions. When coroutine $m$ invokes call$(n)$ we say that coroutine $n$ is attached to $m$ and that $m$ is in a calling state. Similarly the coroutine $k$ is attached to $n$ when it is active.

The two different kinds of control transfer are often used to solve different kinds of problems. The symmetric coroutines can be used to support some kind of quasi-parallel concurrency, where each coroutine represents an independent unit of execution [21, 22]. Although the coroutines execute their code sequentially, it allows one to organize the program into independent sets of execution. In these system the coroutines provide for cooperative multitasking, they have to voluntarily give up control to some central scheduler.

On the other hand we have coroutines that are intended to implement

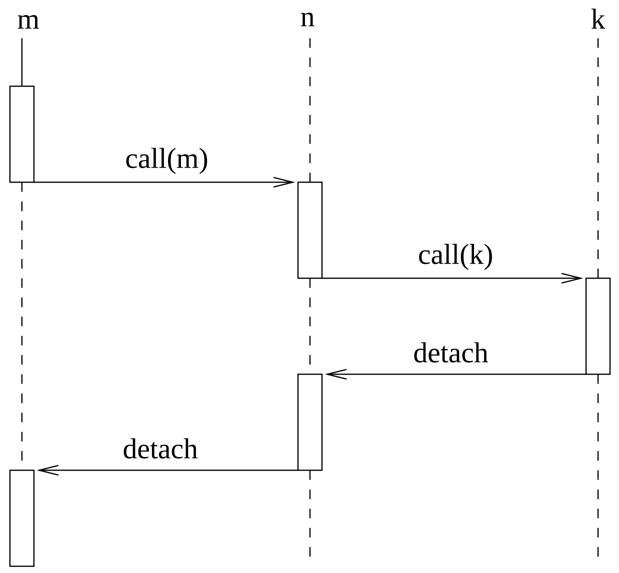

**Figure 2.2:** Asymmetric control transfer mechanism.

constructs that produce some sequence of values, in these scenarios asymmetric control transfer mechanisms can be useful. Examples include *iterators* [33] and *generators* [20, 26].

When symmetric and asymmetric control transfer mechanisms are mixed as they are in Simula, we are left with a powerful programming language construct as we will see in section 2.2.

One primitive operation that we have yet to mention is the one that creates the coroutine, often referred to as simply the create operation. The state that the create primitive leaves the coroutine in varies between the implementations of coroutines. In Simula the newly created coroutine is attached to its creator, letting it run initialization code. If the creator only wanted to create and initialize it, the coroutine must detach itself from its creator explicitly. Other implementations leave coroutines in a detached state after creation as for example coroutines provided by the Lua programming language, keeping the initialization code separate from the body of the coroutine.

### 2.1.2 First-class object or a language construct

The notion of *first-class objects* is usually defined in the context of a particular programming language. Generally we can say that a first-class object is an entity that can be used in programs without restrictions when compared to other kinds of objects in the same language. This can imply that a first-class object can among other things be stored in a variable, be constructed at run-time or be passed as parameters. An exact definition of what it implies that an object is first-class does not exist, it can only be expressed in regard to a particular language.

The coroutines that are *not* first-class objects are those that are constrained as a language constructs. An example of the latter kind is CLU iterators [33], which can only be accessed through the for loop construct, yet they *are* considered coroutines.

Coroutines expressed as first-class objects will have considerable influence on the expressiveness of the programming language. The restricted, non-first-class objects coroutines are typically intended for particular uses and as such they cannot be directly manipulated by the programmer in the same sense.

Most languages provide coroutines as first-class objects, with variable amount of restrictions. Simula class instances and coroutines are exactly the same, thus fulfilling this property to perhaps the highest degree. Other languages provide for a lower level of first-classness, Python generators are also first-class, but this status is inherited from the fact that every method and function in Python is first-class in the sense that they can be passed as parameters and have references to them stored in variables.

### 2.1.3 Stackfullness

Stackful coroutines are able to suspend the execution from within nested functions as well as in the coroutine body. This allows the coroutine to invoke procedures recursively and then suspend the execution of the coroutine somewhere on the resulting stack. The next invocation of the coroutine will continue with the same stack.

Providing stackful coroutines will have great impact on the implementation of the coroutines in the language. This multi-stack behaviour does not fit into all languages, and as such languages such as Python and Perl does not allow stackful coroutine constructs. We see this in section 2.3.2.

Languages that provide stackful coroutines are Simula, Modula-2 and Icon. Of these we will consider Simula coroutines in the next section.

### 2.1.4 Full coroutines

Based on the classification above, Ierusalimschy and de Moura introduces the concept of *full coroutines* as first-class and stackful objects. They argue that symmetric and asymmetric control transfer is equivalent in terms of expressiveness by demonstrating that both can be implemented using the other. Therefore it is sufficient to provide one of the two control transfer mechanisms and still have the coroutines pass as a general-purpose.

## 2.2 Simula coroutines

The Simula programming language was developed in the 1960s at the Norwegian Computer Center, primarily by Ole-Johan Dahl and Kristen Nygaard.

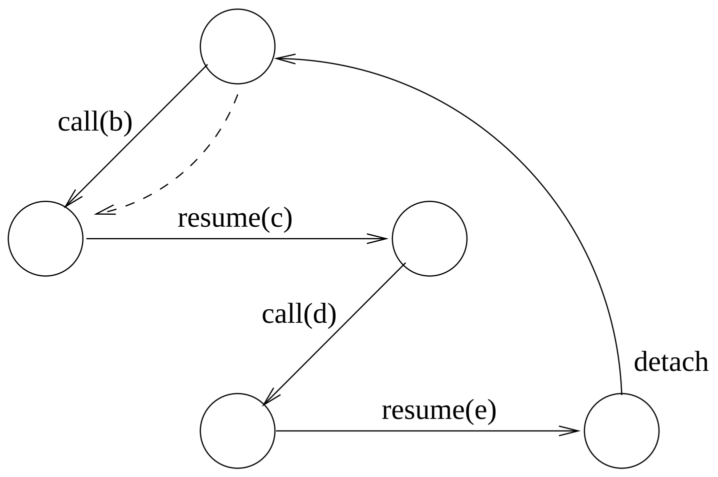

**Figure 2.3:** A dynamic hierarchy of coroutines.

Simula evolved through two main incarnations, SIMULA 1 and Simula 67 [36].

SIMULA 1 was a process-oriented discrete event simulation language based on Algol 60. Simula 67 (or simply Simula), was a general object oriented language, with classes supporting discrete process simulation (the system class Simulation). Simula introduced the object-oriented paradigm and is considered the predecessor of all modern class-oriented languages.

One aspect of the object-model presented by Simula that few of its successors followed, is the ability for objects to act as processes that execute in quasi-parallel. The idea was that classes have code statements that are executed when objects are created, but unlike ordinary subroutines, these can temporarily transfer control to some other object. When control was transfered back, execution continued where it had previously left off. Quasi-parallel sequencing of multiple objects is analogous to the notion of coroutines as Conway described it [8]. Dahl and Nygaard would often refer to the definition by Conway and said that a set of objects functions as coroutines [29].

The coroutines in Simula are first-class since objects act as coroutines when created. Simula coroutines are also stackful, in fact coroutines each have their own stacks and so Simula is often referred to as a multi-stack language. Simula also provide both symmetric and asymmetric control transfer facilities.

By mixing symmetric and asymmetric control transfer facilities, it is possible to build dynamic hierarchies of coroutines such as the one seen in figure 2.3 [34]. The Simula program code that creates such a hierarchy of coroutines can be seen in figure 2.4.

Since Simula coroutines are both stackful and first-class, they are also considered full coroutines in the manner of this discussion.

The names used for the different control transfer mechanisms are still with us today, and the same names are used extensively throughout the rest of this thesis. The operations, resume, detach and call are exactly as those described

```
BEGIN
   REF(A) a; REF(B) b; REF(C) c; REF(D) d; REF(E) e;

   CLASS A; BEGIN Detach; ... Call(b); ... Call(b); ... END;
   CLASS B; BEGIN Detach; ... Resume(c); ... Resume(c); ... END;
   CLASS C; BEGIN Detach; ... Detach; ... Call(d); ... END;
   CLASS D; BEGIN Detach; ... Resume(e); ... END;
   CLASS E; BEGIN Detach; ... Detach; ... END;

   a := NEW A; b := NEW B; c := NEW C; d := NEW D; e := NEW E;
   Resume(a);
END;
```

**Figure 2.4:** The Simula program corresponding to figure 2.3.

above and in Simula these were provided by the Simulation class.

## 2.3 Similar constructs in other languages

Several languages provides coroutines as described above. Some languages also provide for similar constructs that can be used to simulate or implement coroutines. In this section we briefly present a few languages that include coroutines or in some way have constructs that resemble coroutines. We try to compare the features provided with the coroutine characteristics as described above, and when they differ substantially from full coroutines we briefly describe the differences.

### 2.3.1 Lua coroutines

Lua [11] is a lightweight scripting language. It is dynamically typed, interpreted and has automatic memory management facilities with garbage collection. Lua was designed, and is also typically used, as an extension language embedded in host programs written in C/C++.

Lua provides asymmetric coroutines with two basic coroutine primitives, besides create, namely resume (not to be confused with the resume primitive of symmetric coroutines; this Lua primitive corresponds to the one we have called call in the above discussions) and yield (corresponds to detach). An operational semantics for its asymmetric coroutines is described in [12]. Lua coroutines are first class objects, just like ordinary functions.

The Lua creators only provides asymmetric coroutines, arguing that handling the sequencing between symmetric coroutines is non-trivial and that understanding the control-flow in programs employing such can be a considerable effort. They also motivate this decision to preserve easy integration with its host language C.

### 2.3.2 Python generators

Python is an programming language that has gained widespread acceptance the last ten years. It has an extensive object model and it supports several programming paradigms. Several implementations exist, most notably CPython and Jython. Python is dynamically typed, has automatic memory management facilities and functions, methods and classes are first-class objects.

Generator functions [38] were introduced in Python 2.2 with semantics close to coroutines.[1] A Python generator is an ordinary function that contains the yield keyword, but unlike functions, when called, generators returns a *generator-iterator object*. This is an object that conforms to an internal Python iterator protocol, which again can be used in for example for-loops in a convenient way, as seen in figure 2.5. In short this object has a method that on successive calls resumes the generator function body until it reaches a yield statement. Any expression following this statement is returned and the generator is suspended until the next invocation.

```
>>> def gen():
...     n = 0
...     while n < 10:
...         n += 2
...         yield n

>>> [ x for x in gen() ]
[2, 4, 6, 8, 10]
```

**Figure 2.5:** Generator in Python

In other words Python generators are resumable, but unlike ordinary coroutines a Python generator is only able to transfer control back to its immediate caller. A nested yield statement within a nested function will only create a new generator. In this manner Python generators differ in transfer control semantics compared to the other coroutine implementations mentioned above. Python generators are not considered stackful, and so we will not think of them as full coroutines. Python generator semantics is however close to asymmetric coroutines, they provide two operations for invoking the generator; ordinary function invocation and the yield statement.

### 2.3.3 Scheme continuations

Scheme [25] is a functional language, a Lisp dialect, developed by Guy Steele Jr. and Gerald Sussman in the 1970s. The language itself is very simple and minimalistic. Scheme was one of the first languages to support explicit *continuations*.

Understanding Lisp-*closures* is an important part understanding the se-

[1]When this thesis was submitted a proposal to include asymmetric coroutines via enhanced generators [42] was accepted. This will be included in Python 2.5.

mantics of Lisp-like languages and Scheme continuations. A closure is an abstraction representing a function and its environment in which the function was created, making it possible to delay the execution of a function until the closure is invoked, the canonical example being a function that creates an "adder":

```
(define (make-adder n)
   (lambda (x) (+ n x)))
(define add-two (make-adder 2))
(add-two 3)
=> 5
```

**Figure 2.6:** Closures in Scheme

A continuation is a way to represent the computation stack of the program at any given point, in other words a way to represent the rest of a computation. In a way a continuation is a generalization of the normal invoke/return procedure pattern in imperative languages; when a procedure returns to its caller it implicitly invokes the continuation at the point of which the procedure was called. However, with explicit continuations it is possible to invoke *any* continuation, so that a procedure might not return to where it was called from. In this manner a continuation is simply a closure that is explicitly invoked instead of an implicit return when the end of a procedure is reached.

Scheme provides a function call/cc ("call with current continuation") that wraps up the "current continuation" in a first class object similar to a closure and passes it to its argument which must be a function of one argument. If this function invokes the continuation with a value, this value is immediately returned to the continuation of the original call/cc call. However, it is possible to invoke another continuation or function, passing continuations around. Thus in this program: the "rest" of the computation from the view of call/cc is the

```
(+ 1 (call/cc
       (lambda (k)
         (+ 2 (k 3)))))
=> 4
```

**Figure 2.7:** Closures in Scheme

application of (+1...) with the "hole" (the dots) replaced with something. In other words, the continuation of this program is a program that will add 1 to whatever is used to fill ..., and this is what call/cc is called with, bound to k. When the continuation is invoked (as k with an argument it abandons the current continuation and only computes (+ 1 3).

When programming with a continuation-passing style, the previous continuation is passed on every time a function is invoked. This makes it possible to implement many control transfer mechanisms, for example the infamous goto, coroutines, exception-handling and back-tracking to name a few [23].

# Chapter 3

# Java thread model

High thoughts must have a high language.

Aristophanes

This chapter gives a brief summary of the Java thread model and the tools that will be used to build a flexible coroutine framework in Java in chapter 5. Section 3.1 summarizes a few points regarding threads, section 3.2 explains the role of the Java object monitor and 3.3 provides some information on the new concurrency utilities that were introduced in Java 5.0.

The Java programming language was initially developed by Sun Microsystems during the early 1990s. Today the specifications of the language [19], the Java Virtual Machine [32] (JVM) and the Java API are managed through a community lead by Sun, called the Java Community Process. Ever since the beginning, a vital part of the language specification has been the Java memory model, which defines how threads interact through memory. In this chapter we take a close look at the Java thread model, from the low-level details the Java object monitor to the high-level libraries that help us solve parallel programming tasks at different levels of abstractions.

## 3.1 Threads in Java

When we refer to *threads* we usually mean to say threads of execution in which a thread is a sequence of instructions that are executed sequentially. In most programs there is only one thread, they are single-threaded. The opposite of a single-threaded program is a multi-threaded program, that is to say a program with multiple threads of execution of instructions happening simultaneously. This is generally done by either having multiple processors where each thread is executed on separate processors, or by *time-slicing*, where each thread is given some maximum amount of time to execute before another thread is scheduled to run, replacing the first. Threads are often referred to as lightweight processes, threads and processes are similar in many ways, but they differ in the way that threads share resources. Multiple threads within the same process usually share the same memory, but multiple processes on one computer do not.

In Java all threads run within the JVM which allows a program to have multiple threads running simultaneously. The only way to create a new thread is to create an object of the class java.lang.Thread. This new thread is not active, it will not be runnable before its start() method is called. Lea [30] describes the life cycle of a typical thread as seen in figure 3.1. A newly created thread is in the *created* state and calling start() makes the thread *runnable*. Now the fate of the thread is in the hands of the JVM. The JVM may schedule the thread and make it enter its *running* state where it starts to execute its code. After some period of time it may be interrupted by the JVM to let other threads run as well. However it may also execute some code that will cause it to block, for example waiting for a lock or reading from a file, then it will enter the *blocked* state in which it will stay until it is "unblocked". This is taken care of by the JVM in many cases but it may also be unblocked by another thread releasing a lock. A thread is terminated when it returns from its run() method.

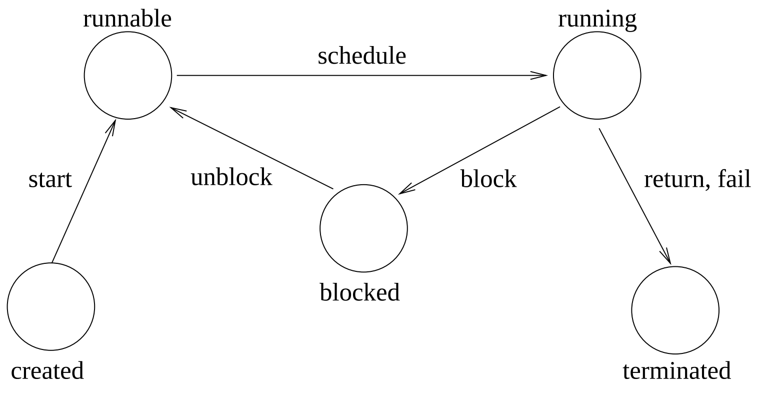

**Figure 3.1:** The life cycle of a Java thread.

In practice there are two different ways of creating a new thread. One way is to create a class that extends java.lang.Thread and then override the run() method. To start executing such a thread, simply invoke its start() method. This new thread will then start executing whatever is in run().

The other approach is to create an object of a class that implements the java.lang.Runnable interface, and then pass this object to a Thread constructor. The Runnable interface declares only one method; run(). In fact Thread implements this interface, but inheriting this class as suggested in the previous paragraph comes with more overhead than just implementing the Runnable interface [1, §9.9]. In many cases it is not necessary to override any other methods than run() in Thread, so a class that implements the Runnable will often suffice.

On multi-processor systems with $N$ processors we generally expect up to $N$ threads to actually execute concurrently, on single-processor systems only one thread may execute on the processor at any given time. Since JVMs may run on many types of systems with varying values of $N$ the Java thread model only gives general policies when it comes to the scheduling of threads. A newly created thread will initially have the same priority as its creator, but this can be changed at run-time.

However it is possible to assign different priorities to different threads and general guidelines are given for how this affects the internal scheduling of threads inside the JVM. As a general rule when there are more runnable threads than processors those with higher priorities will be favoured, but how this happens varies. Lower-priority threads are guaranteed to run only when all higher-priority threads are blocked or terminated, but again how this actually is implemented varies. Some implementations may always choose the threads with higher priorities making lower priority threads wait endlessly, others may mix priorities and aging or other scheduling policies.

Obviously the scheduling of threads with different priorities is not something one can rely on for algorithm correctness, since there are given no promises about the fairness and scheduling policy of the underlying implementation. It can however be used when there are multiple threads with different sets of tasks, e.g. a thread that handles communication with the user by registering mouse clicks should probably have higher priority than a thread that does background computation.

## 3.2 Java object monitors

A monitor is a synchronization mechanism that protects shared data, for example variables that multiple threads need access to. Monitors provide operations that ensure that only one task is *inside*, has access to the shared data, at any given time whilst other must wait outside.

Every object in Java is coupled with an *object lock* which may be held by only one thread at a time, other threads trying to obtain this lock simply have to wait. This lock is not obtained explicitly with a method or keyword, but methods and blocks are declared as protected by this lock by using the `synchronized` keyword.

Similar to the lock that is coupled with every object, there is also an *entry set* and a *wait set* associated with every object and corresponding lock. The wait set holds threads that are blocked on the associated object `o` by calling `o.wait()` and the entry set holds threads that are trying to enter the monitor. Both of these sets are maintained internally by the JVM, and together with the lock this is what makes up the Java object monitor.

The simple class in 3.3 on the next page acts as a monitor, which means that only one thread may execute the code inside its method `f()` at a time. When a thread calls `f()` it first enters the enter set where it tries to acquire the object lock. If it succeeds it enters the monitor and it is said to be the *owner*. When the thread leaves the synchronized block, in this example that happens when it returns from `f()`, the object lock is released and another object waiting to acquire the lock in the entry set now owns the monitor.

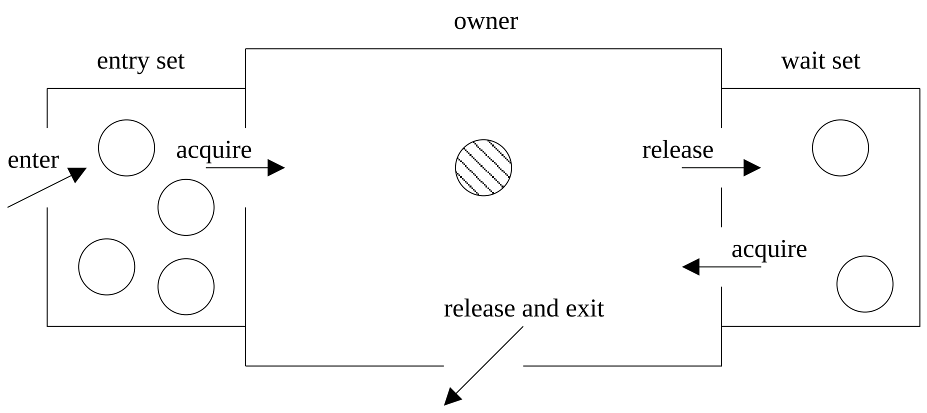

**Figure 3.2:** Java object monitor.

```
class SimpleMonitor {
    synchronized void f() {
        // ...
    }
}
```

**Figure 3.3:** A simple Java class that acts like a monitor

Once inside the monitor a thread may invoke certain monitor operations, these methods are inherited from the Object superclass and these methods may be used to implement various patterns in parallel programming. In fact these methods are the only means of interacting means with the Java object monitor.

**wait()** This method will suspend the calling thread and it will enter the wait set of the target object monitor, atomically releasing the synchronization lock, that is to say thread suspension and object unlocking happens indivisible, no other thread will run while this operation executes. All other locks held by the thread are retained.

**notify()** This method will choose and remove an arbitrarily thread from the wait set and resume it from the point of its wait call. However this thread will have to wait until the notifier releases the synchronization lock. Other threads *may* obtain the lock after the notifier has released it and before the notified is resumed, this will make notified thread block further.

**notifyAll()** This method works in the same way as notify() except that it resumes *all* threads waiting on the lock. This will cause all the threads in the wait set to compete for the owner status once the notifier leaves the monitor.

It is also possible to interrupt a thread in the wait set by calling a method on its thread object (Thread.interrupt()), this will in effect perform the same steps as if the thread was notified except that after the lock is acquired an exception is thrown to indicate that it was interrupted.

## 3.3 Java concurrency utilities

With the release of Java 5.0 a new set of of concurrency utilities was added to the standard Java libraries [31]; java.util.concurrent. The facilities mentioned above has been a part of Java since day one, but they are not always as easy to use as it seems, they operate at lower levels than most programmers would like to descend to. It is possible to write high-level concurrency constructs using the object monitor, such as semaphores, locks and condition variables, but it is also hard to get it right.

The new packages of concurrency utilities include several general purpose synchronization facilities such as semaphores, barriers and exchangers (which allows threads to rendezvous and exchange information). There is also a Lock class which provides multiple wait sets via condition variables.

**locks and condition variables** Ordinary locks that factor out the locks and wait sets of built-in monitors and condition variables that allow for multiple wait sets per lock.

**synchronizers** Classes that help implement common special-case synchronization idioms. Includes a *semaphore* class, which is a classic tool to implement mutual exclusion and resource restrictions. A *barrier* class, i.e. a resettable synchronization point, *latches* to block until a given number of signals, events or conditions hold and *exchangers* that allow threads to exchange objects at a rendezvous point.

**queues** Thread-safe implementations of ordinary FIFO queues.

**executors** Provides an interface for defining custom thread-like subsystems such as thread pools, asynchronous I/O and lightweight task frameworks, and also gives implementations of some common patterns.

**thread-safe variables** Classes that support lock-free and thread-safe programming on single variables and provides atomic operations such as *compareAndSet*.

### 3.3.1 Locks and condition variables

We have already mentioned the Java object monitor. The concurrency utilities added to Java 5.0 also include classes for programming with explicit monitors. This allows multiple wait sets per lock, instead of only one. The Lock and Condition interfaces provides the ability to program with distinct objects. Where a Lock replaces the use of synchronized methods and blocks, Condition replaces the use of the object monitor.

The Lock interface provides methods for explicitly locking and unlocking the lock, as well as other methods for querying the current state of the lock.

The Condition interface basically provide the same functions as the object monitor; means for a thread to suspend execution until another thread notifies it. Using explicit condition variables makes it easier to signal events and conditions.

Figure 3.4 on page 22 has an example of a bounded buffer using a single lock and two condition variables. The lock protects the buffer to achieve mutual exclusion on the shared variables and there are two wait sets. Threads will wait if the buffer is empty when they're trying to perform a get or when the buffer is full and a put is invoked.

### 3.3.2 Semaphores

Semaphores were introduced in 1965 by Dijkstra [13]. In short a semaphore is a protected variable that can only be manipulated by special operations. Semaphores are among the classical synchronization primitives and they can be used to implement mutual exclusion as well as condition synchronization.

A semaphore encapsulates an integer variable, which is always non-negative. The value of this integer can only be manipulated by two *atomic* operations, historically called $V$ and $P$[1] but often referred to as *up* and *down* or *acquire* and *release*. The Semaphore class in java.util.concurrent uses the latter terms and so we will use those in the short introduction to semaphores that follows. Semaphores, or more specifically these operations, are normally implemented as system calls in the operating system, with the operating system briefly disabling interrupts while it is testing the semaphore [39]. Programming languages and their libraries can them build upon these system calls to create their own representations of semaphores or synchronization primitives on a higher abstraction level.

The main differences between an integer and a semaphore are [15]:

1. A semaphore can be given any integer as its initial value, but after it has been created the semaphore can only be manipulated by *incrementing* (*release*) or *decrementing* (*acquire*) it through a given set of operations. The value of the semaphore can generally not be read.

2. Before a semaphore can decrement the semaphore it ensures that the resulting value is positive, and if it cannot decrement it, the thread blocks until it is possible.

The *acquire* operation is used to delay a process until an event has occured. To signal that an event has occured a process can use the *release* operation, which atomically increments the value of the semaphore by one. Conceptually

[1] *V* comes from the Dutch word *verhoog* which means *increase* and *P* comes from the Dutch phrase *probeer te verlagen* which means *try-and-decrease*

we can say that a semaphore counts the number of permits or the number of wake-ups and saves them for future use.

The power of semaphores lies in the fact that *acquire* if decrementing the semaphore results in a negative value. This, combined with the ability to initialize the semaphore with different values gives a powerful construct that can be used to solve a variety of problems.

Semaphores that are initialized to one and used by two or more threads are called *binary semaphores*. Such semaphores are often used to ensure mutual exclusion between the threads. Whenever a thread enters the critical region it *acquire*s the semaphore, thus decrementing it to zero. Decrementing the semaphore again will cause a thread to block until the thread that is already inside the critical region *release*s it.

The Semaphore class that is provided by java.util.concurrent provides all the necessary methods to program with semaphores. It allows us to initialize the semaphore to any integer value and the methods acquire() and release() manipulate the semaphore according to the description above. Other methods are also provided. It is for example possible to query the semaphore for the number of threads that is blocking on it if there are any or even release more than one permits, that is to say increment it by more than one (release(int $permits$)).

```
class BoundedBuffer {
    private Object[] buffer = new Object[n];
    private int count = 0;
    private Lock lock = new ReentrantLock(true);
    private Condition notEmpty = lock.newCondition();
    private Condition notFull = lock.newCondition();

    public Object get() {
        lock.lock();

        try {
            if (count == 0) {
                notEmpty.await();
            }
            Object o = buffer[count];
            buffer[count−−] = null;
            notFull.signal();
            return o;
        } finally {
            lock.unlock();
        }
    }

    public void put(Object o) {
        lock.lock();

        try {
            if (count == buffer.length) {
                notFull.await();
            }
            buffer[++count] = x;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }
}
```

**Figure 3.4:** Bounded buffer using a lock and two condition variables.

# Chapter 4

# Semantics of flexible coroutines

We are getting into semantics again. If we use words, there is a very grave danger they will be misinterpreted.

H.R.Haldeman

This chapter discusses the formal semantics of flexible coroutines as presented by Belsnes and Østvold [2]. Section 4.1 describes the formal process calculus notation that is used when describing the semantics. Section 4.2 presents the basic operations on flexible coroutines. Section 4.3 extends the basic operations to include rules for scheduling and discusses scenarios that will lead to scheduling. Section 4.4 extends the semantics even further to include a local process set. This is built upon in section 4.5 when parameters and return values are added to the semantics. Section 4.6 presents the formal semantics for special flexible coroutines.

## 4.1 Notation

We specify all semantics within the same formal framework, a process-calculus notation with parallel composition of named processes and special operations for communication between processes. One may consider this as a very simple form of process calculus [35].

The *flexible coroutine system* is a parallel composition of multiple named processes where at most one process is active at any given time. A process in a flexible coroutine system is be a thread or a coroutine, and a flexible coroutine system has at least one process and we refer to the processes by names:

$$C_1 \mid \ldots \mid C_n \mid T_1 \mid \ldots \mid T_m \qquad n + m > 0$$

In our formal framework coroutines are named $m, n, \ldots$ and so on and threads are generally referred to as $t$, in some cases with a subscript. The set of actions of a process, in other words its body, is denoted $P, Q, R$ and so on. Using this notation, a flexible coroutine system consisting of a thread process named $t$ with body $P$ and a coroutine process named $n$ with body $Q$ are written as:

$$t\{P\} \mid n\langle Q\rangle$$

The coroutine system often has an active coroutine process, the coroutine that is currently executing,[1] this is marked with double angles like this:

$$n\langle\!\langle P\rangle\!\rangle$$

We use superscript and subscript to denote caller and callee relationships. Superscript denotes caller, upwards the caller chain, and subscript denotes callee. A coroutine named $n$ being the caller of another coroutine $k$ and the callee of $m$ is denoted as:

$$n\langle P\rangle_k^m$$

This is exactly the coroutine $n$ in figure 2.1 on page 7 before coroutine $k$ has detached, i.e. $k$ is still active. A similar notation is used for thread processes, however it does not make sense to talk about the caller of a thread so this is left out. In some contexts the caller or the callee of a coroutine is irrelevant and is left out. Unspecified caller or callee on the right hand side inherits any caller or callee specified on the left hand side. Other times we need to show that there is no caller or callee and we will explicitly mark this with a hyphen like this:

$$n\langle P\rangle_-^k$$

We describe *operations* on coroutines by giving transition rules in the flexible coroutine system:

$$n\langle\!\langle \mathsf{call}(m).P\rangle\!\rangle \mid m\langle Q\rangle \longrightarrow n\langle P\rangle_m \mid m\langle\!\langle Q\rangle\!\rangle^n$$

The operations are invoked in the body of the processes in the system. Often they are invoked on another named process, like in the example above where the $\mathsf{call}$ operation is invoked on $m$ by $n$, resulting in the coroutine system on the right hand side of the arrow. Some rules need to fulfill a pre-condition for it to apply. Similarly some rules express post-conditions that are true when the rule has been applied.

Thread processes may be in an blocking state waiting for a coroutine, these are marked with a $\tilde{t}$. Similarly a coroutine process may be in a waiting state, these are marked with a $\hat{n}$.

In a running program coroutines can easily create a dynamic call hierarchy where a coroutine "at the top" may have a chain of callees. To simplify the writing of these rules we use a compact notation that specifies the chasing of the callee chain:

$$m\langle Q\rangle_{k!}^u \mid k\langle R\rangle$$

[1]We will not distinguish threads as being active or inactive in the same manner. Threads are considered active unless they are blocking.

is short for

$$m_1\langle Q_1\rangle^{u}_{m_2} \mid m_2\langle Q_2\rangle^{m_1}_{m_3} \mid \ldots \mid m_l\langle Q_l\rangle^{m_{l-1}}_{-}$$

where $m = m_1$, $k = m_l$, $Q = Q_1$ and $R = Q_l$. With $u$ we mean either a thread name, a coroutine name or no name at all $(-)$. Here $k$ was denoted *idle* which also implied that $m_l$ was idle. We can also say that $k$ is *waiting*, like this:

$$m\langle Q\rangle^{u}_{k!} \mid \widehat{k}\langle R\rangle$$

This we understand as the obvious, it is $m_l$ that is in the *waiting* state. Similarly for *active* $k$:

$$m\langle Q\rangle^{u}_{k!} \mid k\langle\!\langle R\rangle\!\rangle$$

Sometimes we will not distinguish the different states of $k$ (or rather $m_l$). When we write $k$ with square brackets, we mean that $m_l$ is in one of the above states:

$$m\langle Q\rangle^{u}_{k!} \mid k[R]$$

In section 4.4 we introduce sets of thread processes that belong to coroutine processes, $\phi_n$. We allow for two operations on the set, insertion and removal. We use the following notation to *insert* $t$ into the set $\phi_n$:

$$\phi_n \lhd t$$

Similarly we will have a notation for removing the element $t$ from the set $\phi_n$, we read this as $t$ was taken out of $\phi_n$:

$$\phi_n \rhd t$$

Here $t$ is taken out of the set. These sets are not processes in the system, they only describe the current state of a coroutine. Further discussion of sets can be found in section 4.4.

We use a simple notation for describing substitution of names:

$$P(m/x)$$

With this we understand that all occurences of $x$ in $P$ is substituted with $m$.

We refer to the parameters passed to a coroutine as $\pi$ and the return value of a coroutine as $\rho$.

We also introduce a meta-function that operates on flexible coroutine system configurations and a meta-function that transfers parameters and return values to the bodies of processes. These functions are defined in the text, in sections 4.4.1 and 4.5.1.

## 4.2 Basic operations

The flexible coroutine system consists of coroutine and thread processes. As with classical coroutines, coroutines may perform operations on each other and in this manner they behave just like Simula style coroutines (section 2.2),

| | | |
|---|---|---|
| $u$ | $::=$ | a thread name, a coroutine name or '–'. |
| $n, m$ | $::=$ | coroutine names |
| $s, t$ | $::=$ | thread names |
| $\widehat{n}$ | $::=$ | a waiting coroutine |
| $\widetilde{t}$ | $::=$ | a calling (blocked) thread |
| $k[R]$ | $::=$ | $k\langle R\rangle$ or $\widehat{k}\langle R\rangle$ or $k\langle\langle R\rangle\rangle$ |
| $P$ | $::=$ | $\mathsf{create}(m,.)P'$ or $\mathsf{call}(m).P'$ or $\mathsf{detach}P'$ or $\mathsf{resume}(m).P'$ or $\mathsf{kick}(m).P'$ or $\mathsf{passivate}.P'$ or $\mathsf{yield}.P'$ |
| $C_f$ | $::=$ | $n\langle P\rangle^k_m$ Flexible coroutines |
| $T$ | $::=$ | $t\{\ldots\}$ Threads, may perform only create, call, and kick |

**Figure 4.1:** Formal notation summary.

with a few exceptions. In a flexible coroutine system a thread may perform an operation on a coroutine.[2] We refer to the former as coroutine-coroutine interaction and the latter as thread-coroutine interaction.

As in the classical case, only one coroutine may be active at any given time, but unlike for example Simula coroutines we will not distinguish any coroutine as the *main* coroutine or subprogram. This means that a flexible coroutine system may be idle with no currently executing coroutine and that a *scheduler* manages the execution of the coroutines.

Figure 4.2 on the facing page [2] shows a state diagram for flexible coroutines. The figure shows all state transitions of a coroutine $n$. A solid-line arrow between two states indicate a state transition caused by $n$ performing a coroutine operation, a dashed-line arrow indicate that the transition was caused by another coroutine $m$ performing an operation on $n$. The 'done' transition from *active* to *terminated* is caused by the coroutine $n$ having finished its body or exiting from it abnormally.[3] Figure 4.4 on page 29 [2] summarizes the basic rules described below.

### 4.2.1 Coroutine-coroutine interaction

Coroutine-coroutine interaction is what happens when a coroutine performs an operation on another coroutine. To do this it needs to know the name of the other coroutine, in practice this means that we need a reference to it.

A basic operation is the call operation, which introduces a caller and callee relationship between the coroutines, similar to asymmetric coroutines:

$$n\langle\langle \mathsf{call}(m).P\rangle\rangle \mid m\langle Q\rangle^{-}_{k!} \mid k\langle R\rangle \quad\longrightarrow\quad n\langle P\rangle_m \mid m\langle Q\rangle^n \mid \widehat{k}\langle R\rangle \qquad n \neq m, k \tag{4.4}$$

[2]We will not discuss any further what we mean with thread. We simply think of a thread as a sequence of instructions executing in parallel with other threads. See chapter 3 for a discussion of threads in Java.

[3]Abnormal termination could result from a run-time error or from throwing an exception.

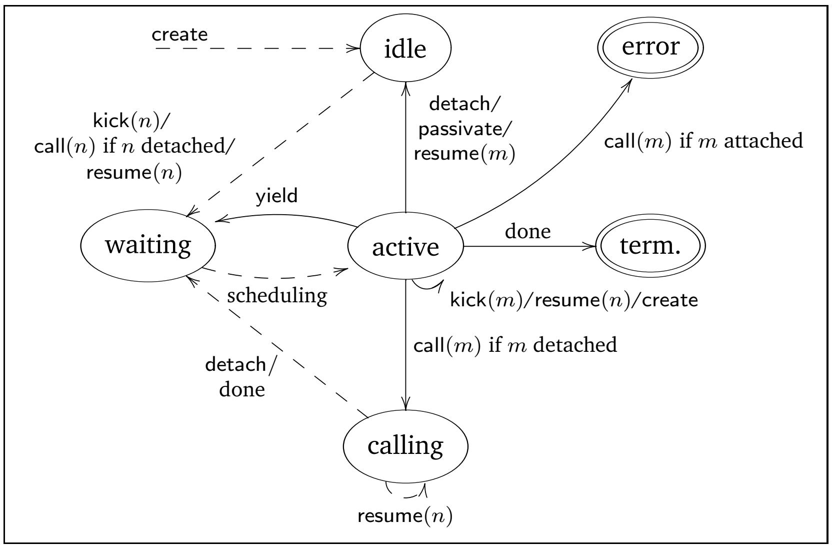
create
idle
error
kick($n$)/ call($n$) if $n$ detached/ resume($n$)
detach/ passivate/ resume($m$)
call($m$) if $m$ attached
yield
waiting
active
done
term.
scheduling
kick($m$)/resume($n$)/create
call($m$) if $m$ detached
detach/ done
calling
resume($n$)

**Figure 4.2:** State diagram for a flexible coroutine $n$.

The intention is to transfer control from $n$ to $m$ and to keep $m$ *attached* to $n$, keeping track of caller and callee. However notice that the callee chain of $m$ is followed and that $k$ is *waiting* and not *active* in the resulting coroutine system. Note that with this semantics the calling coroutine blocks until the callee is not attached anymore.

Both symmetric and asymmetric control transfer operations are provided, just like in Simula. We also provide resume, which transfers control using semantics similar to symmetric coroutines:

$$n\langle\!\langle \mathsf{resume}(m).P\rangle\!\rangle \mid m\langle Q\rangle_{k!} \mid k\langle R\rangle \quad\longrightarrow\quad n\langle P\rangle \mid m\langle Q\rangle \mid \widehat{k}\langle R\rangle \tag{4.7}$$

Just like with the call operation we follow the callee chain of $m$. This is imperative to the execution of the coroutines but it also leads to some complex behaviour. In rule 4.4 we require that $n \neq m, k$, if not there could be a circular reference to the caller in the callee chain.

When a coroutine is attached, as it is after having received a call, it can return to its caller by invoking the detach operation:

$$n\langle P\rangle_m \mid m\langle\!\langle \mathsf{detach}.Q\rangle\!\rangle^n \quad\longrightarrow\quad \widehat{n}\langle P\rangle_- \mid m\langle Q\rangle^- \tag{4.5}$$

Here $m$ is no longer attached to $n$ and control is transfered back to the point, modulo scheduling as we will see later, where $n$ called $m$. It is also possible for a coroutine to detach even if it has no caller, which makes it idle. Notice that when an attached coroutine falls of the end of its body its caller is invoked:

$$n\langle Q\rangle_m \mid m\langle\!\langle 0\rangle\!\rangle^n \longrightarrow \quad \widehat{n}\langle Q\rangle_- \tag{4.2}$$

A new operation in regard to classical coroutines is kick, which is a variant of resume, except that the performer is still active after invocation of the operation.

Very much like an asynchronous resume:

$$n\langle\!\langle \mathsf{kick}(m).P\rangle\!\rangle \mid m\langle Q\rangle_{k!} \mid k\langle R\rangle \quad\longrightarrow\quad n\langle\!\langle P\rangle\!\rangle \mid m\langle Q\rangle \mid \widehat{k}\langle R\rangle \tag{4.8}$$

The operation kick$(m)$ schedules a coroutine $m$ (or the coroutine at the bottom of its callee chain) to be active at some future time.

The passivate operation is similar to detach except that it does not transfer control back to the caller:

$$n\langle\!\langle \mathsf{passivate}.P\rangle\!\rangle \quad\longrightarrow\quad n\langle P\rangle \tag{4.9}$$

The coroutine performing passivate becomes idle and it can be reactivated from the same point later on. If it is attached to a caller this relationship is kept intact.

Lastly when a coroutine $n$ performs the yield operation it signals the scheduler that $n$ may be stopped temporarily and that the scheduler may execute other coroutines now but at some point in the future $n$ wants to become active again:

$$n\langle\!\langle \mathsf{yield}.P\rangle\!\rangle \quad\longrightarrow\quad \widehat{n}\langle P\rangle \tag{4.10}$$

The operations for coroutine-coroutine interaction minus kick, passivate and yield are the same operations as for Simula coroutines with two provisios. First, a scheduler manages the execution of coroutine operations. Second, the lack of a *main* coroutine means that when a detached coroutine performs detach or is done, no special action is taken by the coroutine system.

### 4.2.2 Thread-coroutine interaction (*idle* coroutines)

A thread can interact with idle coroutines in two ways. They may create new coroutines having the same effect as if it was created by another coroutine or they may invoke certain operations on a named coroutine.

A thread $t$ may invoke the call operation on a coroutine $m$:

$$t\{\mathsf{call}(m).P\} \mid m\langle Q\rangle^{-}_{k!} \mid k\langle R\rangle \longrightarrow \quad \widetilde{t}\{P\} \mid m\langle Q\rangle^{t} \mid \widehat{k}\langle R\rangle \tag{4.12}$$

Just like when a coroutine $n$ calls $m$ this will cause $t$ to block until $m$ detaches or is done:

$$\widetilde{t}\{P\} \mid m\langle\!\langle \mathsf{detach}.Q\rangle\!\rangle^{t} \longrightarrow \quad t\{P\} \mid m\langle Q\rangle^{-} \tag{4.14}$$

Alternatively a thread may perform a kick operation on a coroutine $m$ causing $m$ to enter the *waiting* state and keeping $t$ from blocking:

$$t\{\mathsf{kick}(m).P\} \mid m\langle Q\rangle_{k!} \mid k\langle R\rangle \quad\longrightarrow\quad t\{P\} \mid m\langle Q\rangle \mid \widehat{k}\langle R\rangle \tag{4.15}$$

The flexible coroutine system may see a thread as in one of two states, see figure 4.3 [2]. It is either in the state calling which indicates that it is also blocked, i.e. not executing, which it is after it has performed the call operation as in rule 4.12. Or it may be active and executing independently of the coroutine system, and of course any number of threads may be active concurrently.

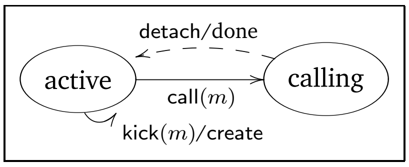

**Figure 4.3:** State diagram for a thread $t$.

Flexible coroutines:

$$
\begin{array}{rlll}
n\langle\!\langle 0\rangle\!\rangle^{-} & \longrightarrow 0 & & (4.1)\\
n\langle Q\rangle_{m} \mid m\langle\!\langle 0\rangle\!\rangle^{n} & \longrightarrow \widehat{n}\langle Q\rangle_{-} & & (4.2)\\
n\langle\!\langle \mathsf{create}(Q,x).P\rangle\!\rangle & \longrightarrow n\langle\!\langle P(m/x)\rangle\!\rangle \mid m\langle Q\rangle^{-}_{-} & m \text{ fresh} & (4.3)\\
n\langle\!\langle \mathsf{call}(m).P\rangle\!\rangle \mid m\langle Q\rangle^{-}_{k!} \mid k\langle R\rangle & \longrightarrow n\langle P\rangle_{m} \mid m\langle Q\rangle^{n} \mid \widehat{k}\langle R\rangle & n \neq m,k & (4.4)\\
n\langle P\rangle_{m} \mid m\langle\!\langle \mathsf{detach}.Q\rangle\!\rangle^{n} & \longrightarrow \widehat{n}\langle P\rangle_{-} \mid m\langle Q\rangle^{-} & & (4.5)\\
n\langle\!\langle \mathsf{detach}.P\rangle\!\rangle^{-} & \longrightarrow n\langle P\rangle & & (4.6)\\
n\langle\!\langle \mathsf{resume}(m).P\rangle\!\rangle \mid m\langle Q\rangle_{k!} \mid k\langle R\rangle & \longrightarrow n\langle P\rangle \mid m\langle Q\rangle \mid \widehat{k}\langle R\rangle & & (4.7)\\
n\langle\!\langle \mathsf{kick}(m).P\rangle\!\rangle \mid m\langle Q\rangle_{k!} \mid k\langle R\rangle & \longrightarrow n\langle\!\langle P\rangle\!\rangle \mid m\langle Q\rangle \mid \widehat{k}\langle R\rangle & & (4.8)\\
n\langle\!\langle \mathsf{passivate}.P\rangle\!\rangle^{m} & \longrightarrow n\langle P\rangle^{m} & & (4.9)\\
n\langle\!\langle \mathsf{yield}.P\rangle\!\rangle & \longrightarrow \widehat{n}\langle P\rangle & & (4.10)
\end{array}
$$

Thread-coroutine interaction on *idle* coroutines:

$$
\begin{array}{rlll}
t\{\mathsf{create}(Q,x).P\} & \longrightarrow t\{P(m/x)\} \mid m\langle Q\rangle^{-}_{-} & m \text{ fresh} & (4.11)\\
t\{\mathsf{call}(m).P\} \mid m\langle Q\rangle^{-}_{k!} \mid k\langle R\rangle & \longrightarrow \widetilde{t}\{P\} \mid m\langle Q\rangle^{t} \mid \widehat{k}\langle R\rangle & & (4.12)\\
\widetilde{t}\{P\} \mid m\langle\!\langle 0\rangle\!\rangle^{t} & \longrightarrow t\{P\} \mid m\langle 0\rangle^{-} & & (4.13)\\
\widetilde{t}\{P\} \mid m\langle\!\langle \mathsf{detach}.Q\rangle\!\rangle^{t} & \longrightarrow t\{P\} \mid m\langle Q\rangle^{-} & & (4.14)\\
t\{\mathsf{kick}(m).P\} \mid m\langle Q\rangle_{k!} \mid k\langle R\rangle & \longrightarrow t\{P\} \mid m\langle Q\rangle \mid \widehat{k}\langle R\rangle & & (4.15)
\end{array}
$$

**Figure 4.4:** Flexible coroutine and thread-interaction semantics.

Notice that unlike in figure 4.2 it is not necessary for $m$ to be detached when the thread performs call. It is of course impossible for a thread to know the state of the coroutine it is about to call. This means that even though $t$ is blocked, the call on $m$ will not be eligible for execution until $m$ detaches.

### 4.2.3 An invariant for active coroutines

From the basic rules we can deduce an invariant over flexible coroutine systems, see figure 4.5.

This invariant says that there will never exist an active coroutine $m$ that has a callee chain $k$, i.e. an active coroutine will never have a callee chain. We can see that this invariant holds for all rules given in figure 4.4. The rule that introduces the caller-callee relationship between $n$ and $m$, the call operation

$$\neg\exists \text{ coroutines } m, k \ : \ m\langle\!\langle Q\rangle\!\rangle_{k!} \mid k\langle R\rangle \qquad\qquad (m \neq k)$$

**Figure 4.5:** A flexible coroutine system invariant.

$$\text{(no active coroutine)} \qquad \widehat{n}\langle P\rangle \longrightarrow n\langle\!\langle P\rangle\!\rangle \qquad (n \text{ scheduled}) \qquad (4.16)$$

**Figure 4.6:** The scheduling rule.

(4.4), ensures that the callee chain of $m$ is chased so that the coroutine $k$ enters the waiting state instead of $m$. Likewise, all rules that transfer control from one coroutine to another, follow the callee chain, e.g. resume (4.7).

A formal proof showing this invariant over the rules in figure 4.4 will not be given, but we use this invariant when we reason about possible configurations and transitions later.

## 4.3 Scheduling

This section introduces a rule for scheduling waiting coroutines and discusses situations that may lead to configurations that are eligible for scheduling.

### 4.3.1 The scheduling rule

So far we have only said that coroutines enter a *waiting* state and that waiting coroutines will be activated at a later time. To ensure that waiting coroutines are activated we introduce a rule that schedules a waiting coroutine to an active coroutine, see figure 4.6.

This simple rule has a pre-condition and a post-condition. The pre-condition is that all the coroutines in the flexible coroutine system is either idle or waiting, i.e. there is no active coroutine. If this pre-condition is true, then some coroutine $n$ is made the activate coroutine.

The post-condition is that $n$ was scheduled. With this we mean that the scheduler picked $n$ out of all the waiting coroutines.

We say that the scheduler is abstract so that we do not attach any restrictions on the formal semantics. However, in an implementation, the scheduler will have to be concrete in some way. Thus it will also enforce a *policy*. The policy of the scheduler is what determines which coroutine should be picked if there are multiple waiting coroutines:

$$\begin{gathered} \widehat{m_1}\langle Q_1\rangle \mid \ldots \mid \widehat{m_i}\langle Q_i\rangle \mid \ldots \mid \widehat{m_k}\langle Q_k\rangle \qquad\qquad k > 0 \\ \longrightarrow \\ \widehat{m_1}\langle Q_1\rangle \mid \ldots \mid \widehat{m_{i-1}}\langle Q_{i-1}\rangle \mid m_i\langle\!\langle Q_i\rangle\!\rangle \mid \widehat{m_{i+1}}\langle Q_{i+1}\rangle \mid \ldots \mid \widehat{m_k}\langle Q_k\rangle \end{gathered}$$

By applying rule 4.16 coroutine $m_i$ is scheduled. It is the policy of the concrete scheduler that picks $m_i$ and not say $m_{i+1}$.

We can imagine a few possible policies. The simplest is perhaps a FIFO queue, the first coroutine that enters the waiting state is the first to be scheduled and so on. However, we can also imagine assigning priorities or even introducing new primitives that affect the scheduling in some way.

### 4.3.2 Transitions that lead to scheduling

The scheduling rule only says that if there is no active coroutine *and* there is at least one waiting coroutine, then a waiting coroutine will be activated. Some of the basic operations will lead to systems that are eligible for scheduling.

From a coroutine-coroutine interaction viewpoint we see that the rules that result in a system that is eligible for scheduling are those where the right hand side does *not* include an active coroutine.

If we study the rules we see that the only rules *with* active coroutines on the right hand side are 4.3 (create) and 4.8 (kick). This implies that all the other rules regarding coroutine-coroutine interaction result in a possible scheduling of a new coroutine, *if* there are any waiting coroutines.

Some rules, such as for example 4.10 (yield) and 4.7 will of course result in at least one waiting coroutine, but 4.9 (passivate) does not. Thus application of the former rules leads to a system that can apply the scheduling rule, the latter does not necessarily do so.

A thread-coroutine interaction view is perhaps more interesting. A thread process cannot possibly know the state of a coroutine, and so thread processes can (and will) invoke operations on coroutines even though the coroutine is not ready to serve the request.

If we study the rules regarding thread-coroutine interaction in figure 4.4 we see that both rule 4.12 (call) and rule 4.15 (kick) result in a waiting coroutine on the right hand side. However none of these rules have active coroutines on their left side. They say nothing about the current state of the coroutine system and it is not possible to deduce from these rules whether there are any active coroutines. Even though they create *waiting* coroutines, we cannot say that these rules leave the coroutine system in a state that is eligible for the scheduling rule since we do not know if there are any *active* coroutines in the configuration.

## 4.4 A local process set

We have formulated an invariant for active coroutines and we have introduced a rule for scheduling a new active coroutine when the coroutine system is idle and has waiting coroutines.

As of now there is a hole in the semantics regarding operations invoked from threads upon coroutines. Threads execute in parallel with the coroutines and they have no knowledge of the state of the coroutines they invoke operations upon. For example when a thread invokes call on a coroutine it can not know if this coroutine is in an state that allows the call to be served immediately. If the coroutine is *active* it can not serve a thread caller until it has finished its current operation. Similarly if the coroutine is in the *waiting* state it is waiting to perform an operation that someone has invoked on it earlier. When a coroutine is not able to serve its caller immediately we say that it is *busy*. The coroutine is also considered busy when a thread invokes call on the coroutine and that coroutine is attached to another process. We want the coroutine to finish serving its current caller.

To fill this hole we introduce rules that for each of the undefined situations and we introduce a *set* that is local to each coroutine $n$ which we name $\phi_n$. This set holds operations from threads that have arrived on the owning coroutine while the coroutine was not able to serve them. The set $\phi_n$ is not a process in the flexible coroutine system, it is only a way for us to describe the current state of the coroutine $n$.

When we add names to the set we will allow for labels to be attached to the name, so instead of saying $\phi_n \lhd t$ we say $\phi_n \lhd t_{\mathsf{call}}$ which says that the operation from $t$ was call.

In the previous section we said that the order in which the scheduling rules were applied on the waiting coroutines was a *policy* that is implemented by the concrete scheduling function. Likewise, we say that it is the policy of the $\rhd$ and $\lhd$ operators on the local process set that determines the order in which threads are serviced, modulo the global scheduling performed by the scheduling rule.

### 4.4.1 A meta-function for chasing the callee chain

Before we define rules that resolve the non-deterministic thread-interaction situations, we define *meta-function* for chasing the callee chain of a particular set of configurations. A meta-function operates on configurations in the transition rules.

The meta-function $\mathcal{K}$ operates on a coroutine $m$ and its callee chain $k!$. The complete definition can be seen in figure 4.7. Later we use $\mathcal{K}$ to avoid writing multiple rules that do basically the same; instead we apply $\mathcal{K}$ where the rules differ. This leads to a smaller rule-set and hopefully simpler semantics.

When $\mathcal{K}$ is applied to a coroutine $m$ and its callee chain $k$ it results in a configuration consisting of the same set of coroutines, $m$ to $k$. When applied on a configuration the state of the coroutine at the bottom of the callee chain $k$ is dependent on the state it previously was in. For example:

$$\mathcal{K}\big(m\langle Q\rangle_{k!} \mid k\langle R\rangle\big) = m\langle Q\rangle_{k!} \mid \widehat{k}\langle R\rangle.$$

The coroutine $k$ is transformed from being *idle* to *waiting*. The rest of the function definition is merely an identity function, *waiting* coroutines are still in the *waiting* state after $\mathcal{K}$ has been applied, likewise for *active* coroutines.

$$\begin{array}{rcl} C & ::= & \text{A flexible coroutine system configuration.} \\ \mathcal{K} & ::= & \text{A callee-chain meta-function.} \end{array}$$

Function domain:

$$\mathcal{K} : C \longrightarrow C$$

Function definition:

$$\mathcal{K}\big(m\langle Q\rangle_{k!} \mid k\langle R\rangle\big) = m\langle Q\rangle_{k!} \mid \widehat{k}\langle R\rangle$$
$$\mathcal{K}\big(m\langle Q\rangle_{k!} \mid \widehat{k}\langle R\rangle\big) = m\langle Q\rangle_{k!} \mid \widehat{k}\langle R\rangle$$
$$\mathcal{K}\big(m\langle Q\rangle_{k!} \mid k\langle\!\langle R\rangle\!\rangle\big) = m\langle Q\rangle_{k!} \mid k\langle\!\langle R\rangle\!\rangle$$

**Figure 4.7:** Callee chain meta-function.

### 4.4.2 Thread-coroutine interaction (*waiting* and *active* coroutines)

As mentioned above, the thread processes will likely invoke operations on busy coroutines. The basic kick rule (4.15) in figure 4.4 mentioned above only say what happens when a thread kicks a coroutine that is idle. Since threads can not know the state of the coroutine systems we also need to define rules for the following situations:

- A thread kicks a coroutine where the coroutine at the bottom of the callee chain is in the *active* state.
- A thread kicks a coroutine where the coroutine at the bottom of the callee chain is in the *waiting* state.

The call rule (4.12) defined above only applies when the callee is not attached to another process, and just like kick it is only defined for *idle* coroutines. That means that the semantics for thread interaction on coroutines also need to include the following situations:

- A thread calls a coroutine where the coroutine at the bottom of the callee chain is in the *active* state.
- A thread calls a coroutine where the coroutine at the bottom of the callee chain is in the *waiting* state.

#### Kicking an *active* or *waiting* coroutine

When a coroutine is *active* it is currently serving some operation. If a thread $t$ invokes kick on coroutine $m$ and $m$ (or the coroutine at the bottom of $m$'s callee chain) is in the *active* state, the request from $t$ must be postponed. When $m$ has finished serving its current operation (with detach), it should continue executing where it left off by serving the pending request from $t$.

The coroutine $m$ will use its local set $\phi_m$ to postpone the request from $t$ when it is busy, and so we can define this behaviour as:

$$t\{\mathsf{kick}(m).P\} \mid m\langle Q\rangle_{k!} \mid k\langle\!\langle R\rangle\!\rangle \longrightarrow t\{P\} \mid m\langle Q\rangle \mid k\langle\!\langle R\rangle\!\rangle \qquad \big(\phi_m \lhd t_{\mathsf{kick}}\big) \tag{4.15$_{\phi_1}$}$$

$$t\{\mathsf{kick}(m).P\} \mid m\langle Q\rangle_{k!} \mid \widehat{k}\langle R\rangle \longrightarrow t\{P\} \mid m\langle Q\rangle \mid \widehat{k}\langle R\rangle \qquad \big(\phi_m \lhd t_{\mathsf{kick}}\big) \tag{4.15$_{\phi_2}$}$$

We see that since $k$ was active the only effect was that we inserted the request from $t$ into the local set of $m$, and that $k$ is still active after the transition.

With these new rules along with the original kick rule, the semantics are clearer. When a thread process invokes kick on an idle coroutine, this coroutine enters the *waiting* state. However, when the coroutine (or the coroutine $k$ at the bottom of the callee chain) is *active* or *waiting* it means that it is either already serving another process or it is waiting to serve another process. We use the local set $\phi_m$ to postpone the request from the thread.

Notice that we insert the request into the local set of $m$, not $k$. When $k$ has finished executing its current operation, this relationship between $m$ and $k$ may be over. Since $t$ invoked kick on $m$, we ensure that it is $m$ that stores the request. It is important to notice this difference; we chase the callee-chain at invocation-time, but when the coroutine is busy we postpone the request on the coroutine that the operation was invoked upon.

This two rules supplement the original kick rule 4.15 from figure 4.4. A new set of rules for thread interaction on coroutines is in figure 4.8.

## Calling an *active* or *waiting* coroutine

We also need to define rules that apply for the call operation from thread processes on a busy coroutine.

We use a similar tactic to define the rules for when a thread invokes call on a busy coroutine. However, there is a second dimension to this operation that we need to take into consideration.

This leads to the following rule, which along with the meta-function $\mathcal{K}$, replaces the original call rule in figure 4.4:

$$t\{\mathsf{call}(m).P\} \mid m\langle Q\rangle_{k!} \mid k[R] \longrightarrow \tilde{t}\{P\} \mid \mathcal{K}\big(m\langle Q\rangle_{k!} \mid k[R]\big) \quad \big(m \neq k, \phi_m \lhd t_{\mathsf{call}}\big) \tag{4.12$_{\phi_1}$}$$

Notice that we label $t$ with call when we add it to the local set of $m$. This allows us to distinguish kick and call operations on the coroutine.

If $m = k$, i.e. if $m$ has no callee chain, and $m$ is *idle*, then the rule above leads to a "double" call on $m$. The $\mathcal{K}$ meta-function sets $m$ in the *waiting* state *and* we add $t_{\mathsf{call}}$ to $\phi_m$. This is the correct procedure if $m \neq k$; we first want $k$ to finish its operation and detach to $m$ so that $m$ is eligible for the attachment rules. We therefore need to define a special rule for when $m = k$ (and notice

| | | |
|---|---|---|
| $\phi_n$ | $::=$ | The local set of a coroutine $n$ |
| $\mathcal{K}$ | $::=$ | The callee chain meta-function |

Thread-coroutine interaction on *waiting* and *active* coroutines:

$$t\{\mathsf{call}(m).P\} \mid m\langle Q\rangle_{k!} \mid k[R] \longrightarrow \tilde{t}\{P\} \mid \mathcal{K}\big(m\langle Q\rangle_{k!} \mid k[R]\big) \qquad \big(m \neq k, \phi_m \lhd t_{\mathsf{call}}\big) \qquad (4.12_{\phi_1})$$

$$t\{\mathsf{call}(m).P\} \mid m[Q]^{-} \longrightarrow \tilde{t}\{P\} \mid m[Q]^{-} \qquad \big(\phi_m \lhd t_{\mathsf{call}}\big) \qquad (4.12_{\phi_2})$$

$$t\{\mathsf{kick}(m).P\} \mid m\langle Q\rangle_{k!} \mid k\langle R\rangle \longrightarrow t\{P\} \mid m\langle Q\rangle \mid \widehat{k}\langle R\rangle \qquad (4.15)$$

$$t\{\mathsf{kick}(m).P\} \mid m\langle Q\rangle_{k!} \mid k\langle\!\langle R\rangle\!\rangle \longrightarrow t\{P\} \mid m\langle Q\rangle \mid k\langle\!\langle R\rangle\!\rangle \qquad \big(\phi_m \lhd t_{\mathsf{kick}}\big) \qquad (4.15_{\phi_1})$$

$$t\{\mathsf{kick}(m).P\} \mid m\langle Q\rangle_{k!} \mid \widehat{k}\langle R\rangle \longrightarrow t\{P\} \mid m\langle Q\rangle \mid \widehat{k}\langle R\rangle \qquad \big(\phi_m \lhd t_{\mathsf{kick}}\big) \qquad (4.15_{\phi_2})$$

**Figure 4.8:** Thread interaction on *waiting* and *active* coroutines.

that $m \neq k$ was stated as a post-condition to the rule above):

$$t\{\mathsf{call}(m).P\} \mid m\langle Q\rangle_{-} \longrightarrow \tilde{t}\{P\} \mid m\langle Q\rangle_{-} \qquad \big(\phi_m \lhd t_{\mathsf{call}}\big) \qquad (4.12_{\phi_2})$$

We need to similar rules for when $m$ is *active* or *waiting*, these are found in figure 4.8, but instead we apply the square bracket notation on $m$ which means any of these three states.

### 4.4.3 The attachment rules

We have defined rules that allow coroutines to postpone requests from threads. The postponed request are added to the coroutine set of outstanding requests. We define an *attachment rule* that removes requests from the local queue and leaves the coroutine in the *waiting* state.

Above we labeled insertions into $\phi_n$ with kick and call. When the coroutine $n$ handles a call operation from $t$ it also needs to be attached to $t$: $n\langle P\rangle^{t}$, the same does hold not for kick operations.

A coroutine $n$ can serve pending operations if it is *idle* and if it is without both caller and callee, i.e. $n\langle P\rangle^{-}_{-}$. When the coroutine is in this condition *and* it has a non-empty local set it is eligible for the attachment rules. If the coroutine $n$ has a call request pending in its local set, then $n$ is attached to the thread that invoked the call and then left in the *waiting* state:

$$\big(t_{\mathsf{call}} \in \phi_n\big) \qquad n\langle P\rangle^{-}_{-} \longrightarrow \widehat{n}\langle P\rangle^{t} \qquad \big(\phi_n \rhd t_{\mathsf{call}}\big) \qquad (4.17)$$

This rule has a pre-condition and a post-condition. The pre-condition is that there is an outstanding call operation from the thread named $t$ in $\phi_n$. If this is true, then the rule can be applied. The post-condition says that it was the name $t$ that was extracted and removed from $\phi_n$.

A similar rule is defined for outstanding kick operations in $\phi_n$. However, kicks are asynchronous and the coroutine should not be attached to the thread that invoked it:

$$\big(t_{\mathsf{kick}} \in \phi_n\big) \qquad n\langle P\rangle^{-}_{-} \longrightarrow \widehat{n}\langle P\rangle \qquad \big(\phi_n \rhd t_{\mathsf{kick}}\big) \qquad (4.18)$$

$$\phi_n \quad ::= \quad \text{The local set of a coroutine } n$$

The attachment rules:

$$\big(t_{\mathsf{call}} \in \phi_n\big) \qquad n\langle P\rangle^{-}_{-} \longrightarrow \widehat{n}\langle P\rangle^{t} \qquad \big(\phi_n \rhd t_{\mathsf{call}}\big) \tag{4.17}$$

$$\big(t_{\mathsf{kick}} \in \phi_n\big) \qquad n\langle P\rangle^{-}_{-} \longrightarrow \widehat{n}\langle P\rangle \qquad \big(\phi_n \rhd t_{\mathsf{kick}}\big) \tag{4.18}$$

**Figure 4.9:** The attachment rules.

The same pre- and post-conditions apply.

It is the policy of the $\rhd$ operator on the local set that determines the order in which label threads are removed from $\phi_n$. We say that this is a policy and not a part of the formal semantics to keep the formal semantics as general as possible. In a concrete implementation a fair policy is a local set that acts as a regular queue, i.e. First-In-First-Out (FIFO). This, coupled with the recommendation of the application of the scheduling rule results in a system that is fair, where coroutines are scheduled in a round-robin manner and serve their callers FIFO.

If $\phi_n$ consists of both $t_{\mathsf{kick}}$ and $t_{\mathsf{call}}$ the configuration is eligible for both rule 4.18 and 4.17. It is determined by the policy of the system which of these have preference, if any. The simplest policy if of course to let the policy of the $\rhd$ operator on $\phi_n$ determine which rule is applicable.

## 4.5 Parameters and return values

To let us communicate with the coroutines in a simple manner we extend the formal semantics to include parameter passing and return values. This enables us to communicate with the coroutines without having to resort to using global variables in an implementation. The idea is to let a process send parameters to coroutines and then having the coroutine return a value, just like an ordinary function call.

The operations provided by asymmetric coroutines, call and detach to attach and detach coroutines, resembles the regular pattern of calling and returning from functions. The idea is to extend these so that parameters are transfered to a coroutine when it attaches itself to another process, either a coroutine or a thread. The caller retrieves the return value when the callee coroutine detaches.

It could also be possible to extend kick and even resume so that they accept parameters, but none of those would allow us to return values back to the caller.

The motivation for adding the local set $\phi_n$ to the semantics was that we wanted to formalize the undefined situations that occur when a thread invokes an operation on a *waiting* or *active* coroutine. This way of postponing requests to coroutines also fits nicely into the way parameters will be passed along to the coroutines.

| | | |
|---|---|---|
| $P$ | $::=$ | The body of a process. |
| $\mathcal{S}$ | $::=$ | The value substitution meta-function. |
| $\pi$ | $::=$ | The parameters to a coroutine. |
| $\rho$ | $::=$ | The return value of a coroutine. |

Function domain:

$$\mathcal{S} : (P, x) \longrightarrow P$$

Function definition:

$\mathcal{S}(P, \pi) =$ substitute all occurences of parameters $\pi$' in P with the new set of parameters $\pi$

$\mathcal{S}(P, \rho) =$ substitute $\rho$ for the occurences of the return value of the coroutine in P

**Figure 4.10:** Parameter and return value substitution meta-function.

Figure 4.10 contains a meta-function that we use to transfer parameters and figure 4.11 contains a summary of the rules presented in this section.

## 4.5.1 A parameter and return value substitution meta-function

We define a meta-function $\mathcal{S}$ for substituting a name with a value in process bodies. This function takes a process body $P$ and a value $\xi$ and substitutes all occurrences of $x$ with $\xi$ in $P$. The function definition is in figure 4.10. Instead of defining this function more formally we say that it just works, i.e. it does the obvious and ensures that parameters and return values are substituted for their correct values.

We use this function to transfer the parameters given to the coroutine to the coroutine body. Since it is also used to transfer return values it needs to operate on bodies of both threads and coroutines.

## 4.5.2 Parameters

As mentioned, we allow the call operation to pass parameters to the callee. When call is invoked on coroutine $n$ with parameters $\pi$ we will denote this as $\mathsf{call}(n, \pi)$. We allow parameters to be passed from both coroutines and threads.

Coroutines can only invoke call on *idle* coroutines. Extending the original coroutine call rule 4.4 is only a matter of extending the left hand side with the parameters $\pi$ and then apply the substitution meta-function on the body of the coroutine on right hand side:

$$n\langle\!\langle \mathsf{call}(m, \pi).P \rangle\!\rangle \mid m\langle Q\rangle^{-}_{k!} \mid k\langle R\rangle \longrightarrow n\langle P\rangle_m \mid m\langle \mathcal{S}(Q, \pi)\rangle^{n} \mid \widehat{k}\langle R\rangle \quad n \neq m, k \tag{4.4$_\pi$}$$

The parameters are passed directly to the callee $m$ using the substitution meta-function and $m$ is attached to $n$. Notice that, even though $m$ was called, it is the coroutine at the bottom of the callee chain that enters the *waiting* state. It is of course the coroutine $m$ that receives the parameters.

We also allow thread processes to invoke call on coroutines with parameters. However, this is not as straightforward as with coroutines invoking call on other coroutines. Remember that we replaced the original call rule 4.12 with the two rules $4.12_{\phi_1}$ and $4.12_{\phi_2}$. We then implied that the attachment rule is responsible for dispatching the call from $t$ to $n$. By dispatching we mean attaching $t$ to $n$ and setting $n$ in the *waiting* state. The scheduling rule then ensures that $n$ is activated. Thus a call operation from a thread onto a coroutine is actually a two-fold process, first the general call rule is applied, then the attachment rule.

We have to acknowledge this when we extend the semantics to include parameters from threads to coroutines. First we redefine rule $4.12_{\phi_1}$ to include parameters:

$$t\{\mathsf{call}(m,\pi).P\} \mid m\langle Q\rangle_{k!} \mid k\langle R\rangle \longrightarrow \tilde{t}\{P\} \mid \mathcal{K}\big(m\langle Q\rangle_{k!} \mid k\langle R\rangle\big) \quad \big(\phi_m \lhd (t_{\mathsf{call}},\pi)\big) \tag{$4.12_\pi$}$$

The only difference between this rule and $4.12_{\phi_1}$ is that $t$ invokes call with $\pi$ and that the post-condition states that the parameters must be passed aside with $t$ into the local set of $m$. Remember that $\mathcal{K}$ chases the callee chain of $m$ and sets the coroutine at the bottom in the *waiting* state if it was *idle*, otherwise it is an identity function.

This means that if $m$ had a callee chain at the time when $t$ invoked its call, then this callee chain is activated. When the chain has detached itself to the point where $m$ is eligible for the attachment rules, $m$ is ready to serve the pending request from $t$ with the parameters. It is imperative for the execution of $m$ that the parameters are not transfered into its body before it has finished executing its current operation.

We see that simply extending the general call rule to include parameters allows the coroutine to be able to have multiple outstanding requests with parameters. The parameters are paired with the caller thus ensuring that each outstanding operation is invoked with the correct set of parameters. This is important if the coroutine returns a value that is dependant on its input, as it ensures that the correct thread is served with the correct set of parameters.

Next we will redefine the attachment rule that dispatches a call operation from the local set of a coroutine:

$$\big((t_{\mathsf{call}},\pi) \in \phi_m\big) \qquad m\langle P\rangle_{-}^{-} \longrightarrow \widehat{m}\langle \mathcal{S}\big(P,\pi\big)\rangle^{t} \qquad \big(\phi_m \rhd (t_{\mathsf{call}},\pi)\big) \tag{$4.17_\pi$}$$

We have altered the pre- and post-condition of the rule so that they include the parameters $\pi$. We also invoke the value substitution meta-function on the body of $m$ with $\pi$ to transfer the parameter values into the coroutine.

| | | |
|---|---|---|
| $\pi$ | $::=$ | Parameters, given to call. |
| $\rho$ | $::=$ | Return value, returned on detach. |
| $\mathcal{K}$ | $::=$ | The callee chain meta-function. |
| $\mathcal{S}$ | $::=$ | The value substitution meta-function. |

Parameter passing:

$$n\langle\!\langle \mathsf{call}(m,\pi).P\rangle\!\rangle \mid m\langle Q\rangle_{\overline{k!}}^{-} \mid k\langle R\rangle \longrightarrow n\langle P\rangle_m \mid m\langle \mathcal{S}(Q,\pi)\rangle^n \mid \widehat{k}\langle R\rangle \quad n \neq m, k \qquad (4.4_\pi)$$

$$t\{\mathsf{call}(m,\pi).P\} \mid m\langle Q\rangle_{k!} \mid k\langle R\rangle \longrightarrow \tilde{t}\{P\} \mid \mathcal{K}\big(m\langle Q\rangle_{k!} \mid k\langle R\rangle\big) \quad \big(\phi_m \lhd (t_{\mathsf{call}},\pi)\big) \qquad (4.12_\pi)$$

$$\big((t_{\mathsf{call}},\pi) \in \phi_m\big) \qquad m\langle P\rangle_{-}^{-} \longrightarrow \widehat{m}\langle \mathcal{S}(P,\pi)\rangle^t \quad \big(\phi_m \rhd (t_{\mathsf{call}},\pi)\big) \qquad (4.17_\pi)$$

Return values:

$$n\langle P\rangle^m \mid m\langle\!\langle \mathsf{detach}_\rho.Q\rangle\!\rangle^n \longrightarrow \widehat{n}\langle \mathcal{S}(P,\rho)\rangle_{-} \mid m\langle Q\rangle^{-} \qquad (4.5_\rho)$$

$$\tilde{t}\{P\} \mid m\langle\!\langle \mathsf{detach}_\rho.Q\rangle\!\rangle^t \longrightarrow t\{\mathcal{S}(P,\rho)\} \mid m\langle Q\rangle^{-} \qquad (4.14_\rho)$$

**Figure 4.11:** Parameter passing and return values.

### 4.5.3 Return values

Just like for ordinary procedure calls the return value of a coroutine may be passed back to the caller when the coroutine detaches. We redefine the original rules for detach. The coroutine may be attached to other coroutines or to threads.

Rule 4.5 says that when an attached coroutine detaches, the caller of the coroutine is left in the *waiting* state. We simply redefine this rule to include return values by applying the substitution meta-function on the body of the callee, with the return value $\rho$:

$$n\langle P\rangle^m \mid m\langle\!\langle \mathsf{detach}_\rho.Q\rangle\!\rangle^n \longrightarrow \widehat{n}\langle \mathcal{S}(P,\rho)\rangle_{-} \mid m\langle Q\rangle^{-} \qquad (4.5_\rho)$$

Notice that detach is label with subscript $\rho$. A similar rule when the callee is a thread process must be defined to replace rule 4.14. Figure 4.11 contains this rule.

## 4.6 Flexible coroutine patterns

While working with flexible coroutines several programming patterns emerged, resulting in coroutines that did different tasks but that shared a set of features large enough to deserve some attention. In this section we highlight two of these and show the formal semantics of these and give a rough description of why and how they can be useful.

In the next chapter we show how these can be implemented using the framework for programming with coroutines that we build and in chapter 6 we show example usage of these concepts.

### 4.6.1 Spinning coroutine

The concept of the *spinning coroutine* is perhaps one of the simplest but also one that has emerged the most times while working with flexible coroutines. The spinning coroutine is simply a coroutine that executes its body in an infinite loop, i.e. it it "spinning" in the sense that it "rotates" when it reaches the end. This constitutes that somewhere in its body it relinquishes control by for example passivating itself or detaching back to a caller, spinning coroutines are rarely active for a long period of time, they more often than not are only active for at most one iteration.

In rules 4.1 and 4.2 we see that when a coroutine reaches the end of its body the coroutine is terminated, following figure 4.2. A spinning coroutine will never reach the *terminated* state. To define the semantics of the spinning coroutine we only need to look at what happens when a coroutine is created and of course what happens when it reaches the end of its body.

We can simplify and say that creating a spinning coroutine is no different than creating a regular coroutine, it is done via a modified version of the create primitive. Creating a spinning coroutine is not much different than creating a regular flexible coroutine, except that the spinning coroutine keeps a copy the original body:

$$\begin{array}{rcll} n\langle\!\langle \mathsf{create}'(Q,x).P\rangle\!\rangle & \longrightarrow & n\langle\!\langle P(m/x)\rangle\!\rangle \mid m_Q\langle Q\rangle^{-}_{-} & m \text{ fresh} \\ t\{\mathsf{create}'(Q,x).P\} & \longrightarrow & t\{P(m/x)\} \mid m_Q\langle Q\rangle^{-}_{-} & m \text{ fresh} \end{array}$$

Then when the spinning coroutine reaches the end of its body instead of terminating itself it starts with a new copy of the original body:

$$n_Q\langle\!\langle 0\rangle\!\rangle \quad\longrightarrow\quad n_Q\langle Q\rangle$$

The case when a terminating coroutine terminates the whole system is not applicable for spinning coroutines. However we need to define what happens when spinning coroutine is attached and reaches the end of its body and we simply say that it still keeps the relationship to its caller:

$$n_Q\langle\!\langle 0\rangle\!\rangle^m \quad\longrightarrow\quad n_Q\langle Q\rangle^m$$

And this is all that needs to be changed to define the concept of a spinning coroutine.

### 4.6.2 The attached-only coroutine

The attached-only coroutine is a bit more subtle than the simple spinning coroutine. The *attached-only* coroutine is a coroutine that only serves callers and that gladly ignores any attempts to have it execute when it is not attached.

We describe the semantics for this special coroutine with a meta-function $\mathcal{F}$ and an extra version of the scheduling rule that schedules attached-only coroutines. We mark these coroutines as $n_C\langle P\rangle$, the subscript $C$ denotes that this is a coroutine of the attached-only type. The modified scheduling rule only

The spinning coroutine:

$$\begin{aligned}
n\langle\!\langle \mathsf{create}'(Q,x).P\rangle\!\rangle &\longrightarrow n\langle\!\langle P(m/x)\rangle\!\rangle \mid m_Q\langle Q\rangle^{-}_{-} && m \text{ fresh}\\
t\{\mathsf{create}'(Q,x).P\} &\longrightarrow t\{P(m/x)\} \mid m_Q\langle Q\rangle^{-}_{-} && m \text{ fresh}\\
n_Q\langle\!\langle 0\rangle\!\rangle &\longrightarrow n_Q\langle Q\rangle\\
n_Q\langle\!\langle 0\rangle\!\rangle^{m} &\longrightarrow n_Q\langle Q\rangle^{m}
\end{aligned}$$

The attached-only coroutine:

$$\mathcal{F}\big(n_C\langle P\rangle\big) = \begin{cases} n_C\langle\!\langle P\rangle\!\rangle & \text{if } n \text{ attached}\\ n_C\langle P\rangle^{-}_{-} & \text{if } n \text{ not attached}\end{cases}$$

$$\text{(no active coroutine)} \qquad \widehat{n_C}\langle P\rangle \longrightarrow \mathcal{F}\big(n_C\langle P\rangle\big) \qquad (n_C \text{ scheduled})$$

**Figure 4.12:** Semantics for special coroutines.

schedules coroutines of this type and the regular scheduling rule is not used for such coroutines.

The meta-function $\mathcal{F}$ takes a single coroutine in the configuration, and depending on whether or not this coroutine is attached or not leaves it in the *active* state or the *idle*. It follows from the definition of this special coroutine that if it has a callee chain then it will always be activated, since the coroutine at the bottom of this chain always has a caller.

The definition of the meta-function as well as the modified version of the scheduling rule can be found in figure 4.12. We apply $\mathcal{F}$ on the right hand side of the scheduling rule, which actually implies that the coroutine *is* scheduled but that it is not *activated* if it has no caller.

Chapter 5

# A flexible coroutine system in Java

Beware of bugs in the code; I have
only proved it correct, not tried it.

Donald Knuth

This chapter presents a flexible coroutine framework written in Java. Section 5.1 gives an overview of the implementation and shows example of usage, sections 5.2, 5.3 and 5.4 gives a detailed presentation of the implementation. Section 5.5 discusses a few issues regarding exception handling in the current implementation and section 5.6 gives the implementation of the special coroutines that were presented at the end of the previous chapter.

There are several issues that arise when implementing such a system in Java, and we discuss the choices taken and outline the effects these have on efficiency, expressiveness and both ease of implementation and ease of use.

Implementing coroutines in Java has been done before. Helsgaun [24] described a Java package for discrete event-simulation based on the facilities provided by Simula programming language and the Simulation class. In his master thesis Borgen [5] implemented a Simula to Java compiler called Jim, that included translating coroutines written in Simula into Java bytecode that could execute in a standard Java run-time environment.

The flexible coroutine framework that we present here will let programmers use flexible coroutines together with threads in their applications so that the sequential nature of coroutines can be combined with parallel application threads. The main intent is therefore to present a useful library for application programmers. Early in the stages of this thesis Java was chosen as the target language and that leaves us with two alternative strategies:

1. Integrate the coroutines as a part of the language, i.e. extend the Java language and either use a compiler-generator such as Polyglot [37] or

extend a Java compiler or a JVM to deal with new constructs.

2. Build the coroutine framework on top of Java, using the facilities provided by the language today such as classes, interfaces and the vast amount of classes in the standard library and provide a standard Java package.

The amount of time and resources available on a project like this is of course limited, which makes the last alternative stand out as the best. However there are better reasons for not selecting the first alternative than just time and resources. We want to provide a framework for using coroutines in Java. Introducing this as a language extension with a new compiler or a pre-processor or perhaps even a new JVM is not what we want. Therefore we implement the coroutines using the standard library and the facilities already in the language. The idea was to let us play with flexible coroutines in real applications and hopefully not keep us spending time on implementing the coroutines themselves. However, as it turned out some of the more subtle areas of the flexible coroutine semantics demanded more attention and in fact a great deal of time was spent on implementing the flexible coroutine framework.

The choice of using standard Java libraries and threads leaves us with two alternatives. We still need to find a way to represent the coroutines in Java and a way to suspend the execution of a coroutine on demand. One way to think of multiple coroutines is to think of them as multiple stacks, where execution control is transfered from one stack to another. Another way is to interleave each of the coroutines on one stack. As mentioned above, Simula coroutines in Java has been done before with two different approaches. The first approach is taken by Borgen, whereas he builds a run-time system on top of Java, thus letting all coroutines run on one stack. The code generated by Jim implements a simple stack and creates objects representing code blocks and maintains both static and dynamic links.[1] The coroutines are then executed in a way similar to how Simula is compiled into C, using multiple named labels[2] for entry and resume points.

Helsgaun takes another approach using threads and creates Java objects that represent the coroutines. Each coroutine is then executed within its own thread. This simplifies a lot as it lets the JVM take care of the stacks and the execution of our coroutines and leaves us with finding a way to control the execution, the suspending and resuming of the threads.

However, every alternative has both positive and negative sides. Whereas the use of threads makes it simple to reason about the implementation, it also comes with a cost. When the number of coroutines increases, the number of threads increases as well. Extensive thread usage will cost more memory. Also more threads means more context switching and more overhead by the JVM and its thread scheduler but as we will see that may not be a problem. Helsgaun shows that one of the most expensive operations is creating and starting a

[1]These links are references and pointers to the surrounding block and the block calling the active procedure respectively. These concepts are shared by most stack oriented languages and their implementations.

[2]Since Java lacks a goto construct this was achieved by using a loop, a global variable determining the current active block and a switch-statement.

thread. He suggests that it might be more effective to reuse the threads as coroutines are finished with them and so he implements a simple thread-pool. Instead of discarding the thread it is returned to the pool and reused by another coroutine. This actually reduces the running time of his examples substantially.

The rest of this chapter presents a Java package for programming with coroutines based on the approach taken by Helsgaun, but without the thread-pool. Adding this thread-pool at a later time will probably be a trivial task, the code that needs to be changed is already quite isolated. Section 5.1 introduces the Coroutine class and shows how it can be used to program with coroutines. It then gives an overview of the implementation of the class, splitting it in three parts; the first part being the code that belongs to the surrounding coroutine system as discussed in 5.2. Section 5.3 shows the code that represents operations that application threads and coroutines invoke on a coroutine. Section 5.4 presents the code that makes up the coroutine object instances. Section 5.5 gives a few notes regarding exception handling involving coroutines and section 5.6 presents a few special coroutines based on the Coroutine class, as discussed in the previous chapter.

The complete source code for the no.nr.coroutines package is listed in appendix A.1. In some cases the code presented in this chapter may differ from the listing, parts that are of no interest have been omitted for brevity and simplicity. The complete code also contains comments, most of these are based on this chapter. Additionally the complete code contains a few helper methods and some code for debugging purposes.

## 5.1 Overview

The Java package presented contains a class called Coroutine that gives programmers a base class for programming with coroutines. This is an *abstract* class that needs to be extended to construct a coroutine:

```
public abstract class Coroutine {
    abstract public void body();
    ⋮
}
```

Since it is abstract it means that it is designed *only* as a superclass and to create a coroutine the programmer needs to create an subclass of it. There is a fine line between an abstract class and an interface, but generally we can say that interfaces *declare* behaviour and that abstract classes *give* behaviour to its subclasses. As we will see below, the Coroutine class gives several private methods to its subclasses as well as an environment for them to live in.

To create a coroutine the programmer needs to create a subclass of the Coroutine class and then provide a method named body(). This method is the body of the coroutine, the code that is executed when it is active. When a coroutine starts executing, for example when it has been kicked by a thread, it starts at the top of this method and executes the statements within it sequentially.

The previous chapter defined semantic rules for operations on coroutines. The Coroutine class provides a set of methods, each corresponding to exactly one coroutine operation. The signatures of these methods are:

**public final static** Object call(Coroutine next, Object params...);
**public final static void** kick(Coroutine next);
**protected final static void** resume(Coroutine next);
**protected final static void** detach();
**protected final static void** passivate();
**protected final static void** yield();

Notice that the public methods coincide with the operations that threads may perform on coroutines, in addition to instance creation. The methods declared as protected coincide with the operations coroutines perform on each other or themselves, meaning that they can be called inside any subclass of Coroutine, but not outside or by any unrelated classes. All methods are final meaning that they cannot be overridden by a subclass. They are also declared as static, this means that a thread invokes the methods through the class, for example invoking Coroutine.call($c$) to call coroutine $c$. The signature of the call() method deserves a few extra words. Its return type is an Object reference, this is the return value of the coroutine and it also takes an unknown number of Object references.[3] Not all coroutines take parameters and not all return values. If a coroutine wants to declare that it returns a value or that it is parametrized it must implement the following interfaces:

**interface** Parameterized { ... }
**interface** Returning { ... }

The details of these will be shown later. The intent however is to let a coroutine define its behaviour as parameterized and/or returning. Notice that none of these constructs will create a *type-safe* coroutine. It is possible to give parameters of the wrong type or even fail to give parameters to a coroutine that expects them. We will discuss this in more detail in section 5.5.

### 5.1.1 A simple example

Having outlined the Coroutine class and the interface given to application programmers it can be useful to see a simple example of usage before we dive into the implementation details. Figure 5.1 shows two simple coroutines as static inner classes of the MyExample class.[4] The main method calls the coroutine $m$ which prints “1” and then calls $n$. $n$ is called with a parameter $p_0$ and $n$ prints $p_0 + 1$ and then returns $p_0 + 2$. When $n$ detaches back to $m$, $m$ prints the return value and then *resumes* $n$ again, this time without a parameter. When $n$ is resumed it prints $p_0 + 3$ and then falls of the end of its body. As

[3]The ellipsis after the formal parameter params declare that this is a *vararg* method, it takes a variable number of arguments and packs them into an array of Object references. These arguments may also be of primitive types, such as int, which are *auto-boxed* into corresponding wrapper classes. These constructs were added to Java 5.

[4]The body() methods are marked with *Override*. This is a called a *method annotation* and was introduced in Java 5.0. It does not alter the semantics of the code but it is used by the programmer to show that she intends to override a method in a superclass.

a consequence $m$ is left in its idle state and the coroutine system is idle. The resulting output of this example is 1234.

```
class MyExample {
    static Coroutine m = new M();
    static Coroutine n = new N();

    static class M extends Coroutine {
        @Override
        void body() {
            System.out.println("1");
            r = (Integer) call(n, 1);
            System.out.println(r.toString());
            resume(n);
        }
    }

    static class N extends Coroutine implements Returning, Parameterized {
        @Override
        void body() {
            System.out.println(parameter0 + 1);
            returnValue = parameter0 + 2;
            detach();
            System.out.println(parameter0 + 3);
        }
    }

    public static void main(String[] args) {
        Coroutine.call(m);
    }
}
```

**Figure 5.1:** Example usage of the Coroutine class.

### 5.1.2 Implementation overview

Even though the Coroutine class is one entity, it can be useful to think of it as comprised of three parts, tightly bound together on both instance and class level. Throughout the rest of this chapter we will use the following terms over and over again:

**operations** The coroutine operations are the methods coinciding with the various operations presented in the previous chapter. Some of these are declared as public and exported outside the class, like kick() and call(), others are only visible to the subclasses of Coroutine. Each is implemented as a single Java method, together the set of these methods comprise the part of the code known as the *operations*.

**system** The parts that comprise the coroutine *system* are the wait queue and the scheduler. The scheduler employs a dispatcher. These parts are internal to the Coroutine class.

**instances** The instances of the Coroutine class inherit methods and fields that are bound to the instance, for example the coroutine thread and methods for suspending and resuming this thread.

These parts are discussed in the following sections. However since they are all related to each other the choices made in one place will have consequences. It can be difficult to find a specific order in which to discuss these parts, and often it will be necessary to reference methods or fields that have not yet been discussed, however this will be limited to the absolute minimum.

Before we deal with these individual parts it can be useful to have a minimum knowledge of how they fit together and how coroutines are suspended and resumed.

As mentioned above the idea is to let each coroutine execute within its own thread. When the application programmer creates a coroutine and activates it from a thread, using either kick() or call() the *coroutine runner thread* is instantiated and the coroutine starts executing its body. This runner thread is exclusive to each coroutine, it is in this thread that the coroutine executes its body. We will refer to this thread as the runner thread or the coroutine thread, other threads, that is to say those that do not belong to a coroutine body will be referred to as *application threads*. Sometimes we will simply use *threads*, but hopefully it will be clear from the context whether we mean application thread or coroutine runner thread. Throughout the discussion we also use the general term *process* to mean a process participating in a flexible coroutine system, i.e. either an application thread or a flexible coroutine.

If the coroutine does something that in effect suspends it, for example invoking detach(), we want to suspend the runner thread belonging to the coroutine in question. It is necessary to understand *where* these methods are called, that is to say which thread execute the different methods. When the coroutine is activated for the first time, let us say that it was done with kick() this time for simplicity, every step up until the actual starting of the coroutine runner thread with thread.start() is executed in the thread that activated the coroutine. This is what we mean with *dispatching a coroutine*. We will also use the word dispatching when we reactivate a coroutine.

When start() returns, the stack belonging to the application thread is now somewhere inside the methods of the Coroutine class and it falls back down into its own code after returning from the various methods it has called. Continuing our example, after the coroutine has done some work it calls detach(). This method is then executed in the runner thread of the detaching coroutine.

We have now come to the point in which the coroutine needs to suspend its own thread. Throughout the implementation we use multiple binary semaphores, initialized to zero, some of these are created dynamically:

- Each coroutine has a semaphore. This semaphore is used to suspend the coroutine runner thread, we will refer to it as the coroutine semaphore.

- For every call() operation from an application thread to a coroutine, a binary semaphore is created. This semaphore is used to suspend the calling application thread, and it is named threadBlockingSemaphore.

To suspend the runner thread the coroutine tries to acquire its semaphore that has an initial value of zero. In section 3.3.2 we saw that acquiring a semaphore of zero means that the thread is blocked waiting for someone to release it. This means that this thread blocks and it will not unblock until someone releases its semaphore. If an application thread at a later time invokes kick() on this coroutine then the code inside the Coroutine class will see that this coroutine has been previously active, and still executing in this application thread, the semaphore blocking the runner thread of the coroutine is released.

This was a rough overview of how the threads and coroutine resume control of each other, coroutines blocks on semaphores that are initially zero and other threads (either application threads or runner threads) increase this semaphore. Whenever we have multiple threads referring to the same objects, in this case the semaphores in the runner threads and the coroutines, there is a chance of both deadlocks and race conditions. The race conditions are eliminated by carefully executing the code in mutual exclusion to other threads. By ensuring that a thread that holds this mutex sooner or later will release it we also eliminate the deadlocks.

## 5.2 Coroutine system

We start by showing the parts that comprise the *coroutine system*. In the current implementation the coroutine system is equal to the static namespace, in other words there can only be one coroutine system in a program. Refactoring the code to support multiple coroutine systems is possible. The system can be encapsulated in objects of for example a CoroutineSystem class. Some effort needs to be spent on refactoring the coroutine instances so that they can be bound to a single system. In section 6.2 we discuss the need for multiple systems.

The methods and fields that make up the system-wide parts of the Coroutine class are approximately as seen in figure 5.2. Notice that the fields and the methods are both private and static, meaning they are accessible to all instances of the class, but not its subclasses.

The systemQueue is simply a linked list of coroutines. This list holds references to the waiting coroutines in the coroutine system. We will often refer to this list as simply the *queue*. There is also a reference to the currently active coroutine in systemCurrent. If the system is idle it points to null. The most important of the three variables is systemLock. This lock is necessary to guarantee that only one thread at a time executes inside the schedule() method.

```
abstract class Coroutine {
    private static Lock systemLock;
    private static Coroutine systemCurrent;
    private static Queue<Coroutine> systemQueue;

    private static void schedule(boolean forced) {
        // ...
    }
}
```

**Figure 5.2:** Implementation of the flexible coroutine system.

### 5.2.1 System lock

When we say that the system queue and the scheduler are system wide we also imply that they are shared and in a multi-threaded system it is important to avoid concurrent use of shared resources. When a thread invokes an operation on a coroutine, most of the code is actually executed in this application thread, not in a coroutine thread. To protect the coroutine system from having multiple threads manipulate the same objects at once we employ a *lock* called systemLock. Each thread that wishes to manipulate any variable that belongs to the coroutine system needs to hold this lock before it continues.

Normally such blocks of critical code follow the syntactic blocks and in such situations a simple synchronized code block is sufficient. However in this code, more often than not, the critical region span multiple methods and the start point and end point are not necessarily in the same method block.

Figure 5.3 shows two threads that almost simultaneously try to invoke an operation on two different coroutines. Solid lines mean that the thread is executing, dashed lines mean that it is idle, in this case it is blocked. The thread $t_1$ invokes kick$(m)$ and then grabs the system lock. While $t_1$ is executing the kick() method the thread $t_2$ invokes kick$(n)$. However when it does so $t_1$ is somewhere deep inside the coroutine system and so $t_1$ holds the system lock which causes $t_2$ to block. Then when $t_1$ returns from kick it releases the system lock and at this moment $t_2$ grabs it and at this point it executes the body of kick$(n)$.

### 5.2.2 Scheduler

The scheduler is responsible for activating a waiting coroutine whenever this is deemed necessary as described by the formal semantics in the previous chapter. In the previous chapter we said that the policy of the scheduling was determined by the concrete scheduler, the abstract scheduling in the formal semantics laid no restrictions on the order of which waiting coroutines are served.

The concrete scheduler in this implementation is represented by the schedule() method and the systemQueue. The former method selects the next

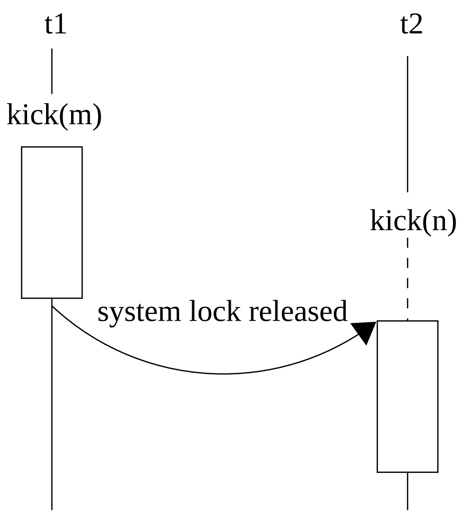

**Figure 5.3:** Two threads competing for the system lock.

coroutine from the systemQueue depending on the context and activates it. Coroutines enter the systemQueue as soon as they enter the *waiting* state, and we use systemQueue as a simple FIFO queue, the first coroutine to enter the queue is the first to be scheduled. This coincides with the simplest of the recommendations in the previous chapter. In section 6.1 we discuss the possibility of having other kinds of scheduling policies.

We described the scheduling rule as a simple rule that ensures that whenever there are waiting coroutines *and* no active coroutine, a waiting coroutine is scheduled to run. In section 4.3 we saw that some rules lead to configurations that were eligible for scheduling, while others did not. Most importantly we saw that we could tell whether or not to schedule after the thread-interaction rules had been applied. By describing the semantics of scheduling in a multi-coroutine system we saw that there where a multitude of different scenarios in which the scheduler was invoked, sometimes it was necessary for it to activate a new coroutine, other times not, all depending on the context and on the number of waiting coroutines.

The implementation uses two different ways of invoking the scheduler and refers to these as *forced* and *non-forced* scheduling. The difference between these may seem subtle but it is important. With forced scheduling the schedule() method will *always* select a new coroutine if the queue is non-empty and then activate it. In other words the caller is forcing the scheduler to activate a new coroutine. However with non-forced scheduling the schedule() method will only activate a new coroutine if there is no active coroutine at the moment, which means that it will activate a new coroutine if and only if systemCurrent is null. Table 5.4 shows which method invokes the scheduler with the forced flag set and unset, both from a coroutine-coroutine-interaction and thread-coroutine-interaction viewpoint.

In the thread-coroutine column the scheduler is invoked in a non-forced

| *Operation* | *Coroutine-coroutine* | *Thread-coroutine* |
|---|---|---|
| resume | forced | — |
| detach | forced | — |
| yield | forced | — |
| passivate | forced | — |
| call | forced | non-forced |
| kick | non-forced | non-forced |

**Figure 5.4:** Forced and non-forced scheduling of coroutines.

manner in both kick and call. This is because the thread invoking the operation (which is also the thread that will execute the schedule() method) has no knowledge of whether or not there is an active coroutine at the moment of its invocation.

When a method invokes the scheduler it cannot know whether the system queue is empty or not.[5] If the queue is empty when the scheduler is invoked then there are no coroutines to activate and the effect should be that the current coroutine should be suspended. The leave() method takes care of this by suspending the runner thread of the current coroutine. We will take a close look at this method, as well as its sibling, the enter() method in section 5.4.

It can be useful to see what this important piece of code looks like. Figure 5.5 shows the code copied directly from the complete source code modulo the comments. Notice that this concrete scheduler has not much in common with the scheduling rule of section 4.3. The concrete scheduler is in fact more than this method and the system queue, it also consists of the coroutines *entering* the system queue. The scheduling rule was meant to be applied when the coroutine was waiting and coroutines enter the *waiting* state by applying the attachment rules. In this implementation this happens in the detach() method, that we discuss in section 5.3.6.

The three last lines of the method simply pull the next waiting coroutine from the wait queue and activates it, using the enter() method. This method will resume the next coroutine by releasing its semaphore, figure fig : passivateandenter shows how the two threads (the thread executing the schedule() method and the runner thread of the coroutine that should be activated) actually execute in parallel for a short period of time.

## 5.3 Coroutine operations

Next we will see how the various methods that coincide with the coroutine operations are implemented. The signatures for these methods were given above; here we see how they are implemented. It is important to keep in mind that kick() and call() can be invoked by application threads. This will be important when we consider who is responsible for releasing the system lock.

[5]In *some* cases it can, e.g. the yield() method inserts the current coroutine into the system queue and thus the system queue will at least contain this coroutine.

```
private static void schedule(boolean forced) {
    if (!forced && systemCurrent != null) {
        // get out of here and let caller release systemLock
        return;
    }

    // if the queue is empty then the whole system should be put to
    // sleep, but remember we own the systemLock!
    if (systemQueue.isEmpty()) {
        Coroutine coroutine = systemCurrent;
        systemCurrent = null;
        // leave() will release systemLock
        coroutine.leave();
        // and when we're awakened we just return
        return;
    }

    Coroutine next = systemQueue.poll();
    next.state = State.ACTIVE;
    next.enter();
}
```

**Figure 5.5:** Implementation of schedule() method.

Most of these methods, with the exception of call() and detach(), are in fact quite simple. They all manipulate either the active coroutine, through the systemCurrent reference, or the "next" coroutine, either given as an argument to the method or found as the callee of the active coroutine in the case of detach().

The various operations on coroutines will be presented in order of increasing complexity. We start of with the simplest of them all, passivate and yield and then move on to the more complex operations.

### 5.3.1 The Operation class

Before we dive into the various coroutine operations and their implementations we need to present an important class that is used throughout the code. The Operation class is a private inner class in the Coroutine class and it is used extensively to represent operations that are performed on or by a coroutine.

The following shows the most important fields of this class:

```
private static class Operation {
    OperationType operationType;
    Coroutine coroutineCaller;
    Thread threadCaller;
    Semaphore threadBlockingSemaphore;

    Object[] parameters;
    Object returnValue;
    ⋮
}
```

The previous chapter introduced the local set $\phi_n$, that hold references to threads that invoked operations on the coroutine $n$ while $n$ was not able to handle any new requests. In this implementation, the instance variable localQueue corresponds directly to the local set $\phi_n$. We will refer to this as the local queue of a coroutine and its name, queue, implies that it has the same FIFO policy as the system queue.

In the formal semantics we inserted references to threads (labeled with the kind of operation) into $\phi_n$ in for example rules $4.12_{\phi_1}$ and $4.15_{\phi_1}$. The implementation does the same, but we wrap these references in Operation objects. We generalize it even further and also use such objects when coroutines invoke call on other coroutines. These Operation objects also hold the parameters that should be passed to the coroutine when the request is invoked, and the return value that must be passed from one thread to another is stored here.

A short example demonstrates how we use the Operation class. When a coroutine $n$ calls another coroutine $m$, an instance of the Operation class is created. In the case of coroutine-coroutine call, the formal semantics transfers the parameters immediately. The same happens in the implementation. However, a reference to this operation object is also stored in the coroutine. If $m$ had been invoked by a thread instead of the coroutine $n$, the formal semantics demand that the thread and its parameters are added to the local queue, this is exactly what happens in the call() method that we present in section 5.3.5.

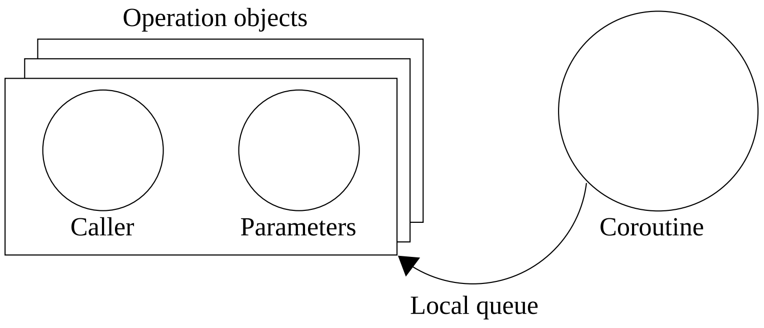

**Figure 5.6:** Encapsulation of caller and parameters in the local queue of a coroutine.

Another important responsibility of the Operation instances is to store the parameters that callers pass to coroutines. Remember that a coroutine can have several outstanding calls from different threads with different sets of parameters. Figure 5.6 shows the relationship between a coroutine and the Operation objects; each coroutine has a local queue that keeps multiple objects which encapsulate the caller and its parameters.

In addition to references to the caller and the parameters, the Operation objects also keep a reference to a Semaphore object called threadBlockingSemaphore. This semaphore is created when a thread calls a coroutine and it is initialized to zero. The call operation is synchronous from the threads point of view, meaning that the thread that invokes it should block until the coroutine detaches from *that* request. This semaphore is used to block the thread in the call() method. The callee releases this semaphore when it is done serving this thread. We will study this in more detail in section 5.3.5.

## 5.3.2 Passivate

The simplest method is passivate(), so we start by showing how it is implemented and then discuss why it is so and give a detailed description of what happens when a coroutine invokes it.

The passivate() method can only be invoked by coroutines meaning that it always executes in the coroutine instance thread, or the runner thread as we also refer to it as, and never in regular application threads. The semantics is quite simple: put the coroutine to sleep. This implies that we should try to invoke the scheduling rule, i.e. if any other coroutines are waiting let one of them run. The complete code for passivate() is only three lines:

```
protected final static void passivate() {
    systemLock.lock();
    systemCurrent.state = State.IDLE;
    schedule(true);
}
```

Notice that the method grabs the system lock, to make sure that the statements that follow execute in mutual exclusion to other threads. This is done because we do not want to invoke the scheduler in more than one thread at at time, doing so could lead to a race between the different threads. Although the coroutines execute in mutual exclusion the application threads do not.

Although the code may seem simple, let us take a closer look at what it does and how it effects a system with multiple coroutines. Imagine a coroutine $m$ invoking passivate and that a coroutine $n$ is in the *waiting* state and at the front of the system queue. This coroutine $n$ has previously invoked resume($k$) and is now blocked. In figure 5.7 we see how control is transfered. In this figure there are two threads, the runner threads of the two coroutines $m$ and $n$. A solid line means that the thread is executing, i.e., the coroutine is active and a dashed line means that the thread is inactive.

When $m$ invokes passivate() it grabs the system lock and then calls the scheduler. Since $n$ is waiting in the system queue the scheduler calls $m$.enter(),

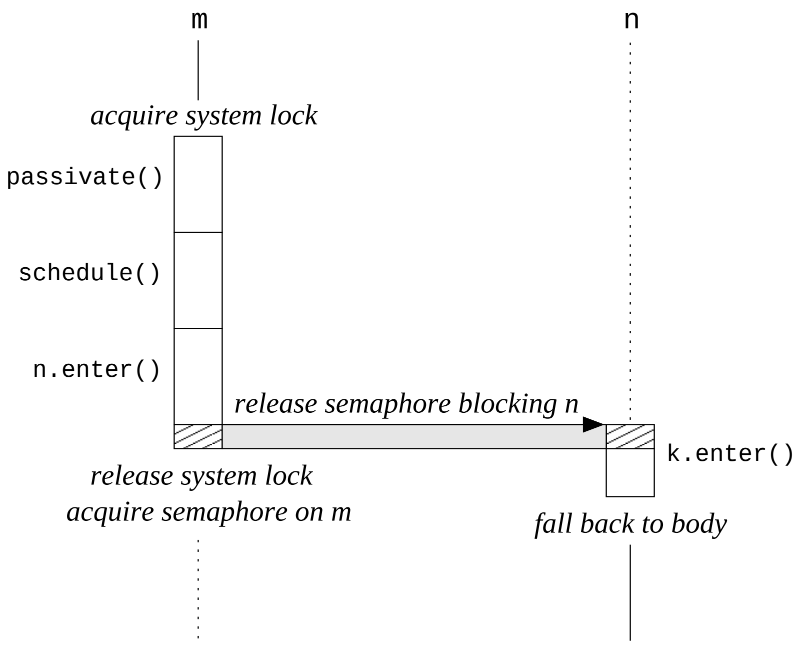

**Figure 5.7:** Passivating $m$ awakens waiting coroutine $n$.

still executing in the coroutine thread of $m$. Here the semaphore that is blocking $n$ is released awakening $n$. For a slight period of time the two runner threads execute in parallel, marked with grey in the figure. Finally the runner thread of $m$ tries to acquire the semaphore on $m$, which is zero, causing the runner thread to block just like $n$ had done previously. Just before it does this it releases the system lock.

We said that $n$ had previously invoked resume($k$), which means that its thread will acquire the semaphore on $n$ (since $m$ released it) still inside $k$.enter(). From here the runner thread of $m$ will fall back to its instance of the body() method and continue where it left of, following the resume($k$) statement. As we see the enter method is an important piece of code and it is one of the key methods that delivers control from one coroutine to the next. We will study this method in section 5.4.

However if the queue is *empty* when $m$ passivates, the scheduler acts differently. In that case it will invoke $m$.leave() and the system lock will be released in *this* method before $m$ tries to acquire its semaphore.

Clearly, the code that is executed while holding the system lock is not necessarily composed of a single syntactic block. It is still possible to use synchronized blocks, but since it is not possible to know the exact sequence of methods invoked, it is harder and quite possible doing so would lead to code that is hard to read. Using explicit and non-lexical locking, instead of the synchronized blocks makes this part of the code easier to understand.

*Every* method discussed below grabs the system lock and then updates the state of the coroutine according to figure 4.2. From now on we will briefly mention the state change but not necessarily show the code that updates the state field.

### 5.3.3 Yield

The next primitive we consider is the yield operation. This method is a way for the coroutine to relinquish control momentarily, we can think of it as a signal or a hint to the scheduler that it could let other coroutines run now but that this coroutine wants to be active again at a later time. Just like passivate this operation can only be invoked by coroutines, meaning that it always executes inside a runner thread.

Like passivate(), this method is reasonably short, most of the work is done in the scheduler. The complete code that makes up the yield() method is four lines:

```
protected final static void yield() {
    systemLock.lock();
    systemCurrent.state = State.WAITING;
    systemQueue.add(systemcurrent);
    schedule(true);
}
```

Whereas the passivate() method only updated the state of the current coroutine and then invoked the scheduler, this method also ensures that the coroutine enters the system queue. Since this method can only be invoked by coroutines that are in *active* state, it is guaranteed that the coroutine is not already in the system queue (since the coroutine cannot be in two states at the same time).

However it *is* possible that there are outstanding requests on the coroutine that invoked yield. In that case, these request are in the local queue of the coroutine. Figure 5.8 shows this relationship between the system queue and the local queue. The top row shows the system queue containing two coroutines $m$ and $n$ and each of these has a local queue which again holds multiple operations. Note that the system queue cannot hold more than *one* instance of each coroutine; either the coroutine is in the system queue or it is not.

If we return to the yield() method we see that after it has added the coroutine to the system queue it invokes schedule($true$). This means that we *force* the scheduler to activate a new coroutine, just like in the case of passivate(). However when passivate() invoked the scheduler it could not be sure that there was any waiting coroutines, the queue could be empty. In this case we are guaranteed that there is at least one coroutine in the system queue, namely the one we just inserted. So either is this call to schedule() redundant, because the next coroutine is the current one and its next operation is this yield operation, or it is necessary because some other coroutine is waiting first in the queue.

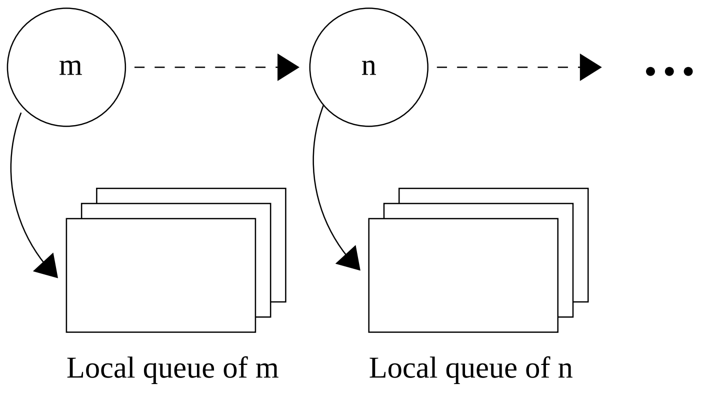

**Figure 5.8:** System queue and local queue relationship.

### 5.3.4 Kick

The next primitive operation that we will consider is the kick() method, which is an asynchronous way to activate a coroutine. This is the first of the primitives that we consider that also can be invoked by application threads. By now we have actually covered most of the details behind the system queue and the local queue as well as the Operation class. Before we analyze the details of the kick() method let us take a look at the code:

```
public final static void kick(Coroutine next) {
    systemLock.lock();

    if (!(Thread.currentThread() instanceof Coroutine.Runner))
        // invoked by an application thread
        next.localQueue.add(new Operation(OperationType.KICK));

    // chase callee chain
    while (next.callee != null)
        next = next.callee;

    // apply meta-function K
    if (!systemQueue.contains(next) && !next.isActive())
        systemQueue.add(next);

    schedule(false);
    systemLock.unlock();
}
```

This method is a bit more complicated than the methods presented above, but as we see it follows the semantics of the kick() rules. A labeled operation object is added to the local queue of the coroutine, then its callee chain is chased, and depending on the status of the coroutine $k$ at the bottom of this chain, $k$ is added to the system queue.

Earlier methods only grabbed the system lock, and some other method was responsible for releasing it. This method is the first that locks and unlocks the system lock in the same scope. At first sight this may seem wrong, because we have already determined that $m$.enter() is responsible for unlocking. However if we proceed with this pattern in this method we introduce a subtle bug.

To see why, we need to distinguish between the two different modes of invocation of this method. Let us first consider what happens if a thread $t$ invokes kick() on a coroutine $m$. First the system lock is grabbed by $t$ and then following the formal semantics an Operation object is added to the local queue of $m$. Second, still in the thread $t$ we invoke schedule($false$) . This means that the coroutine system will only activate a new coroutine if there is no active coroutine at this time.

Deep down the scheduler will or will not resume the runner thread of the coroutine in question. If there is an active coroutine it means that it should not activate a new one, however this means that it does not reach the enter() method where it was supposed to release the system lock. In this case, schedule() returns almost immediately to kick() with $t$ still owning the system lock. The only way to ensure that it is unlocked is to unlock it at this point. If there is no active coroutine a special case in the enter() method will ensure that the system lock is *not* unlocked in that scope, implying that whoever called enter() is responsible for unlocking the system lock.

If a coroutine invokes kick() on another coroutine, it also starts by grabbing the system lock. The important aspect is that we invoke the scheduler as *non-forced*, that is to say we pass $false$ as the argument. Obviously there is an active coroutine and thus this invocation of the scheduler will return almost immediately, since systemCurrent is non-null. And before we fall back to the body of the coroutine, we must release the system lock.

### 5.3.5 Call

Moving up on the complexity scale, the next primitive operation method that we will discuss is call(). This method is significantly more complex than the ones we have presented so far. To make sure we cover every aspect of this method we will need to split it in two; the first part discusses coroutine-coroutine interaction and the second part thread-coroutine interaction.

Figure 5.9 shows first part of the complete call() method, handling the case when it is invoked by another coroutine. Figure 5.10 shows the second part of the same method, this time the else-clause that handles the case when it is invoked by an application thread.

The complexity of this method is due to several points:

- It can be invoked by application threads and coroutines and their callers should block. However this is implemented in different ways, the suspending of coroutines is inherently a part of the underlying system via the enter() method, but the suspension of application threads is not.
- Callers can pass parameters to the callee, however these cannot be passed to the coroutine immediately if it was invoked by an application thread due to the uncertainty of whether there are any other calls pending.

- Coroutines can return values to their callers when they detach(). However for the caller these values will need to come when call() method returns.

Since much of the complexity comes from the parameters and the return values, we will first show how these are stored in the coroutine system and then passed around. Then we take a closer look at the two different invocation modes, coroutine-coroutine and thread-coroutine.

### Parameters and return values in Operation objects

Recalling the Operation class from section 5.3.1, we see that there are two important fields in its declaration: private static class Operation Object[] parameters; Object returnValue; When a process invokes call() on a coroutine, an instance of the Operation class is created. If this coroutine implements the Parameterized interface then an array of Object references is passed into call().[6]

The formal semantics for parameters and return values in figure 4.11 shows that in the case of a coroutine-coroutine call the parameters are passed along to the coroutine immediately. If a thread-coroutine call was invoked, the parameters are stored in $\phi_n$ along with a reference to the caller. In that case the parameters are transfered to the coroutine through the attachment rule.

When the coroutine detaches with a return value the modified detachment rules of figure 4.11 imply that the return value should be passed to the caller immediately. In the implementation, the caller and the callee execute in two different threads. To pass information from the callee thread (the coroutine runner thread) to the callers thread we use the Operation object. When the caller awakens in the call() method it fetches the return value from the Operation object.

As we see, the instances of the Operation class also serve as placeholders for parameters and return values. The system lock ensures that only one thread manipulates these objects at a time.

### Coroutine caller

We first consider the situation where the process that invoked call() is a coroutine, so called coroutine-coroutine interaction. This code is shown in figure 5.9.

To find out whether the caller is a coroutine or an application thread, we check the type of the current thread. If the code executes in a thread that is an instance of the Coroutine.Runner class, then surely this must be a coroutine caller.[7]

For the sake of this presentation, let us refer to the caller (the coroutine that invoked call()) as $n$, and the callee as $m$.

[6]See section 5.7 for a discussion regarding the type-safety of the current implementation.
[7]This inner class is private and generally not accessible to the outside.

```
public final static Object call(Coroutine next, Object... params) {
    systemLock.lock();
    if (Thread.currentThread() instanceof Coroutine.Runner) {
        op = new Operation(OperationType.CALL, systemCurrent);
        next.operation = op;
        next.caller = systemCurrent;

        if (next instanceof Parameterized) {
            ((Parameterized)next).setUpParameters(params);
        }

        // chase callee chain
        while (next.callee != null) next = next.callee;

        systemQueue.add(next);
        schedule(true);

        if (next instanceof Returning) { retval = op.getReturnValue(); }
        else { retval = null; }
    } else {
        // thread-coroutine interaction
    }
    return retval;
}
```

**Figure 5.9:** Implementation of call operation part 1.

The formal semantics demand that the coroutine $m$ is *idle*, i.e. not *waiting*. To invoke a coroutine-call on a non-idle coroutine is an error, and that part of the code has been omitted.

As we see, the method follows the formal semantics by transferring the parameters immediately. If coroutine $m$ has a callee chain, then this is chased and the coroutine $k$ at the bottom is added to the system queue. The formal semantics say nothing of whether or not $k$ is allowed to be *waiting* and we demand that it is *idle*.

The runner thread of $n$ will block somewhere inside the enter() method of the next activated coroutine, which was invoked by schedule(). Remember that the next activated coroutine is not necessarily $m$, there can be other coroutines in the system queue. However sooner or later, $m$ is activated and at this point $n$ is still blocked. When $m$ detaches it awakens its caller as we see in the next section, that is to say $n$ enters the wait queue. When $n$ is picked by the scheduler and then made active by the dispatcher it resumes execution inside the enter() method where it was blocked, and eventually falls back to the statement succeeding call().

Following the point in call() where the scheduler was invoked we see that the method checks to see if its callee $m$ supports returning a value. If it does, then this is fetched from the Operation object. This return value is placed in this operation object and *not* in the coroutine $m$. Coroutine $m$ may have served numerous callers since it detached from $n$ and thus it is possible that it has returned a multitude of different values since then. This return value is then returned to $n$. Notice that if $m$ does not support return values, then null is returned. This is a weakness in the implementation that we discuss in section 5.7

#### Thread caller

Next let us look at call() from a thread-coroutine interaction view. This code is shown in figure 5.10.

Most of the code is quite similar to the one that solves coroutine-coroutine interaction, however there are two distinctions.

1. Threads do not block in the enter() method.
2. The invocation of call() may come at a time where the coroutine system is serving a coroutine, be it the callee of this invocation or another, or that the callee is not available to serve this caller at this moment.

The first problem is solved by adding a semaphore to the Operation instance if this is a thread-coroutine call() . This semaphore is initialized to zero. The thread tries to acquire this semaphore after it has returned from schedule() which of course causes it to block. When the callee detaches it releases this semaphore which signals that the calling thread can continue.

Remember that with thread-coroutine call() , it is the attachment rule that transfers the parameters. In this implementation we transfer the parameters immediately *if* the coroutine is in a state that supports it, as seen in the else-clause of the first if-statement. Otherwise we follow the formal semantics and queue the parameters along with a reference to the thread in the local queue of the callee.

As before, we chase the callee chain and add the coroutine found at the bottom to the system queue. However, with thread-coroutine interaction we allow for the thread to invoke call() on the coroutine even though it is busy, so we only add the coroutine to the system queue if it is not already there.

### 5.3.6 Detach

This section presents the last of the complex operations on coroutines, detach. Remember that detach is the asymmetric counterpart to call: a coroutine invokes detach() when it has finished serving a caller.

This method has three responsibilities:

1. Reactivate the caller, either another coroutine or an application thread.

```
public final static Object call(Coroutine next, Object... params) {
    systemLock.lock();
    if (Thread.currentThread() instanceof Coroutine.Runner) {
        // coroutine-coroutine interaction
    } else {
        op = new Operation(OperationType.CALL, Thread.currentThread());
        boolean sched = false;

        if (next.operation != null) {
            next.localQueue.add(op);
            if (next instanceof Parameterized)
                op.setParameters(params);
        } else {
            next.operation = op;
            sched = true;
            if (next instanceof Parameterized) {
                ((Parameterized)next).setUpParameters(params);
            }
        }

        while (next.callee != null) next = next.callee;
        if (!systemQueue.contains(next)) systemQueue.add(next);
        if (sched) schedule(false);

        systemLock.unlock();

        try {
            // block the calling thread (i.e. this thread)
            op.threadBlockingSemaphore.acquire();
            if (next instanceof Returning)
                retval = op.getReturnValue();
            else retval = null;
        } catch (InterruptedException ie) {
            // ...
    } }
    return retval;
}
```

**Figure 5.10:** Implementation of call operation part 2.

2. Transfer return values from the detaching coroutine to the caller, if applicable.

3. Ensure that the coroutine is reactivated if there are pending operations on it by applying the attachment rule.

Notice that we check whether there *is* an Operation object all the time. This is because it is allowed for coroutines to invoke detach() even though they where not attached. The effect is the same as a coroutine that invokes passivate() when it has no caller.

```
protected final static void detach() {
    systemLock.lock();

    if (systemCurrent instanceof Returning && systemCurrent.operation != null) {
        systemCurrent.operation.setReturnValue(
            ((Returning)systemCurrent).getReturnValue());
    }

    if (systemCurrent.operation != null &&
        systemCurrent.operation.isCoroutineCall()) {
        systemQueue.add(systemCurrent.caller);
        systemCurrent.caller.callee = null;
        systemCurrent.caller = null;

    } else if (systemCurrent.operation != null &&
               systemCurrent.operation.isThreadCall()) {
        systemCurrent.operation.threadBlockingSemaphore.release();
    } // else; we have no caller so don't do anything special

    systemCurrent.operation = null;

    if (!systemCurrent.localQueue.isEmpty()) {
        Operation op = systemCurrent.localQueue.poll();
        if (systemCurrent instanceof Parameterized) {
            ((Parameterized)systemCurrent)
                .setUpParameters(op.getParameters());
        }

        systemCurrent.operation = op;
        if (op.isCoroutineCall())
            systemCurrent.callee = op.coroutineCaller;
        systemQueue.add(systemCurrent);
    }
    schedule(true);
}
```

**Figure 5.11:** Implementation of detach operation.

If the caller was a thread, then this thread is now trying to acquire a semaphore that was initialized to zero. The detaching coroutine simply releases this semaphore and then the thread caller can continue in the call() method by picking up the return value. If there is no caller, that is if the operation that invoked the detaching coroutine was neither a coroutine call nor a thread call, then no action is taken.

To transfer return values back to an possible caller, we use the Operation object that was created when the coroutine was called. Remember that the detach() method executes in the runner thread of the detaching coroutine, and we want the return value to be transfered to the thread that invoked call on this coroutine, either an application thread or another coroutine runner thread.

The last responsibility is to ensure that the coroutine is reactivated if another process has a pending operation on it. This is also known as the attachment rule. It follows that if the local queue of the coroutine s non-empty then there is at least one pending operation and if so detach() adds the detaching coroutine to the system queue.

**Summary**

This concludes our presentation of the coroutine operations. The resume() method can be found in appendix A.1.

## 5.4 Coroutine instances

Last we consider the parts that we think of as the coroutine instances. We have seen that coroutines execute their code in a thread that is bound to each coroutine instance, and we have referred to this as the *runner thread*. This section presents methods that the scheduler uses to suspend and resume the runner threads. Two important methods, enter() and leave() have only been briefly mentioned above, and we have seen that the former is invoked by the scheduler when it wants to activate the coroutine and that the latter is invoked if the scheduler wants to suspend a coroutine without activating another.

### 5.4.1 The coroutine runner thread

The coroutine runner thread is a private inner class in Coroutine that extends the standard Thread class. Its only responsibility is to execute the body() method of a coroutine. Remember that threads start executing whatever is in their run() method, and when they fall of the end of this method they terminate.

The run() method of the Runner class is quite simple:

```
public void run() {
    coroutine.body();
    coroutine.state = TERMINATED;
    detach();
}
```

We see that falling of the end of the coroutine body is implemented as a detach operation that leaves the coroutine in the *terminated* state. The coroutine field is a reference to the coroutine that owns the runner thread.

Usually one activates a thread by invoking its start() method, however we find that we often need to restart it over and over again. The go() method in the Runner class hides the difference of starting and restarting a thread:

```
public void go() {
    if (!isAlive()) {
        start();
    } else {
        coroutine.semaphore.release();
    }
}
```

Notice that to reactivate the thread the go() method signals a semaphore in the coroutine.

### 5.4.2 Suspending and resuming coroutines

The ability to suspend and resume the runner thread of a coroutine is an important aspect. It is what makes is possible to implement coroutines in the first place.

There are two methods that deals with this on the instance level; enter() and leave(). They both manipulate a semaphore that is bound to the coroutine. Each coroutine has a semaphore field that is simply called semaphore. This semaphore is initialized to zero and as we have seen, it acts as a binary semaphore.

#### The enter() method

The complete code for the enter() method can be seen in figure 5.12. The main responsibility for this method is to activate the runner thread of the coroutine that it is called upon. If this is the first time this coroutine is activated it also needs to instantiate a Runner object for the coroutine to execute in. If we where to extend the system to use thread pools as suggested by Helsgaun [24] we would have to change this so that it grabs an already instantiated thread from the pool.

The method does different things depending on whether there already is an active coroutine or not. If there is no active coroutine, i.e. systemCurrent is null, then this coroutine has been activated by an application thread. In that case, it resumes the runner thread by invoking go() and then returns. The rest of the method deals with the case where systemCurrent is non-opnull.

When systemCurrent is non-null it means that the method is executing inside a coroutine runner thread. This might seem like a bold statement but it is bound to be true. An application thread never invokes the scheduler as *forced* (see figure 5.4), so if systemCurrent is non-null the application threads depart the schedule method before the dispatcher is called (see figure 5.5). If we are in a runner thread it is necessary to suspend the currently active coroutine before activating the coroutine bound to this method. First it invokes go(), then carries on with unlocking the system lock. At this point there are *two* active runner threads; the current thread that executes the enter() method and the runner thread that was just (re)activated. However the current thread will not carry

```
private void enter() {
    if (runner == null) runner = new Runner(this);
    if (systemCurrent == null) {
        systemCurrent = this;
        systemCurrent.runner.go();
        return;
    }

    Coroutine previous = systemCurrent;
    systemCurrent = this;

    systemCurrent.runner.go();
    systemLock.unlock();

    if (previous.isTerminated()) {
        return;
    }

    try {
        previous.semaphore.acquire();
    } catch (InterruptedException ie) {
        // ...
    }
}
```

**Figure 5.12:** The enter() method.

on for a long time, because next it acquires the coroutine semaphore which is zero, which causes it to block.

Remember that we are still executing in the runner thread of the previously active coroutine. If this previously active coroutine is finished, or as we say *terminated*, we need to ensure that its runner thread dies as well (or is given back to a thread pool). Before we lock down the runner thread we ensure that the coroutine not terminated.

The importance of the coroutine semaphore should not be underestimated. Its initial value of zero will cause the previously active coroutine to block here. Say we have some application thread that tries to activate a coroutine $n$. The runner thread of coroutine $n$ is also alive, but not active, i.e. it has not been scheduled by the JVM. When the runner thread of $n$ gets picked, its next move is to acquire its semaphore (which is zero) so that it blocks. If the JVM lets the application thread execute for a long time before it schedules $n$, this thread will release the semaphore even though there is no one blocking on it. When the runner thread of $n$ gets picked by the JVM and reaches the acquire-statement the value of the semaphore is now one, and the runner thread will pass it. We see that the semaphore lets us be flexible when handling invocations on coroutines.

### The leave() method

The complete code for the leave() method can be seen in figure 5.13. This method is responsible for suspending a coroutine runner thread, and nothing else. It is only invoked if the active coroutine invokes the scheduler when the queue is empty. The active coroutine then tries to acquire its own semaphore, which causes it to block.

```
private void leave() {
    systemCurrent = null;
    systemLock.unlock();

    state = State.IDLE;

    try {
        semaphore.acquire();
    } catch (InterruptedException ie) {
        // ...
    }
}
```

**Figure 5.13:** The leave() method.

## 5.5 Exception handling

This section discusses a few issues regarding exception handling in the implementation presented above. Exception handling is eminently missing from the current implementation, mostly because this was not considered as important as finishing the implementation with parameters and return values.

Essentially Java provides two types of exceptions, *checked* and *unchecked*. In short, unchecked exceptions are objects of classes that have the RuntimeException class in their inheritance path and checked exceptions are objects of classes that do not. The key difference is however that checked exceptions *must* be handled by the programmer to avoid a compile-time error, and the programmer does this by either catching them (using the try – catch construct) or by throwing the exception further up the stack. The latter is done by adding a throws clause to the method signature. This means that every checked exception that is not caught must be explicitly thrown.[8] Java follows an exception-handler model that is referred to as *terminating* [43], which means that if an exception travels up the stack without getting caught, the thread eventually terminates.

Exceptions are a vital part of complex systems, and application programmers will probably find it necessary to employ some exception handling in the coroutine body. The current implementation is free of exception handling, so

[8]Besides being tedious, this leads to awkward bugs involving the versioning of APIs. Because of this, the designers of C♯ removed this distinction and made all exceptions unchecked.

we would need to extend it a great deal if we want to throw exceptions in the coroutine body.

The multi-threaded nature of this implementation makes exception handling harder than it might actually seem. In the current implementation, if an exception is thrown in the body of a coroutine (that is either directly or indirectly in the body() method), that exception would travel up the stack of the runner thread belonging to that coroutine. Apart from the fact that there is no throws-clause in the signature of body(), that coroutine would eventually end up at the top of that runner thread stack, in other words somewhere inside the coroutine system code and here it would fall of the stack and thus terminate the runner thread of the coroutine that threw the exception. The consequences of this is severe, if the runner thread dies then the coroutine essentially ceases to exist, the next time it gets scheduled *that* thread (the thread that executes the scheduler) will get an exception due to the illegal state of the runner thread.

Clearly, if we want to throw exceptions in the coroutine body we need some better protection.

We see that there are two basic problems regarding exception handling in the current implementation:

1. Coroutine can throw both checked and unchecked exceptions. An exception that falls of the runner thread stack will terminate the thread.

2. Exceptions thrown in the coroutine body propagate up the stack of the runner thread. If the coroutine was invoked by an application thread, the application thread will never see the exception.

The first problem can be solved. To prevent unchecked exceptions from terminating the coroutine runner thread we could add a catch-clause at the top of the runner thread stack that catches all types of exceptions[9], and then clean up any garbage before setting the coroutine in the *terminated* state. Checked exceptions *could* also be solved in this manner, but we would also have to carefully add throws-clauses to every method that could participate in such a stack, and then catch these at the top of the stack, for example in the run() method.

However the second problem is more complex. When an application programmer throws an exception in the body of a coroutine, the intent seen from the flexible coroutine framework viewpoint is unclear.

The coroutines can be invoked by application threads either synchronously via call() or asynchronously via kick(). Intuitively if an exception happens when an application thread has synchronously invoked the coroutine, we would like for that exception to propagate through call() and then get caught somewhere in the application thread. But as we have seen, exceptions propagate upwards the stack it was thrown in, not across stacks. This problem is also solvable, but it would be much harder to solve in a clean manner. It *is* possible to catch the

[9]The Exception class is the superclass of all checked exceptions and RuntimeException is the superclass of all unchecked exceptions.

exception and then re-throw it in another thread, but to implement this in the current framework we would need a better way of signalling that an error has occured than what is currently in place.

Implementing this signalling and re-throwing the exceptions is of course also possible, but then another problem arises; what strategy should be chosen for asynchronously invoked coroutines? Obviously it is not possible to re-throw the exception up the application thread stack, since the application has continued after invoking the coroutine. On the other hand, this is not optimal; we would like for the application thread to be notified that an exception occured when the coroutine executed. The solution to this problem is not straightforward, and further discussion is future work.

## 5.6 Flexible coroutine patterns

In section 4.6 we mentioned that whilst programming with flexible coroutines some patterns emerged. In this section we show how these can be implemented in a simple manner by using the framework we have built.

These special case coroutines are implemented as subclasses of the Coroutine class. Unfortunately this is not an optimal solution since Java disallows multiple inheritance.[10] This means that a class can only extend one of the classes presented below. If it wants to implement more than one pattern, it is only possible to inherit the functionality of one of them.

A better approach would have been to let the Coroutine class and the subclasses below implement the *decorator pattern* as described by Gamma et al [18]. This pattern lets you attach additional behaviour to objects in a simple manner. This can be seen in the Java I/O libraries, where functionality is wrapped around an object using constructors.[11]

### 5.6.1 Spinning coroutine

The simplest pattern is that of a coroutine that simply loops forever, the *spinning coroutine*. Section 4.6.1 presents the formal semantics for this coroutine. Coroutines of this kind never enter the *terminated* state and they never fall of the end of their body, instead they start all over at the top of their body until the end of time.

Implementing a spinning coroutine is in fact very simple. According to the formal semantics all we need to do is to restart the body of the coroutine when it reaches the end, in other words we simply let it execute in an infinite loop.

[10]However, even though multiple inheritance could help us here, it would make it hard to implement the pattern coroutines in a correct fashion.

[11]This allows you to customize the input streams, an example usage of this can be a FileInputStream instance that is decorated with a BufferedInputStream and a LineNumberInputStream: new LineNumberInputStream(new BufferedInputStream(new FileReader(filename))). This chain of constructors result in an InputStream object that is buffered and with the ability to use line numbers on the data.

The complete code for the spinning coroutine pattern is therefore only a few lines as can be seen in figure 5.14.

```
abstract class SpinningCoroutine extends Coroutine {

    @Override
    void body() {
        while (true) {
            spinningBody();
        }
    }

    abstract void spinningBody();
}
```

**Figure 5.14:** The SpinningCoroutine pattern.

## 5.6.2 The attached-only coroutine

The attached-only coroutine, as presented with formal semantics in section 4.6.2, is a coroutine that only executes its body if it is attached to another process, be it a thread or a coroutine. The code that implements this coroutine is shown in figure 5.15.

```
abstract class AttachedOnlyCoroutine extends Coroutine {
    Coroutine c = new Coroutine() {
        public void body() { attachedBody(); }
    };

    void body() {
        while (true) {
            if (this.caller() != null)
                call(c);
            detach();
        }
    }

    abstract void attachedBody();
}
```

**Figure 5.15:** The AttachedOnlyCoroutine pattern.

It is a bit more complicated than the SpinningCoroutine that was presented in the previous section. Programmers that wish to implement use this class need to implement the abstract attachedBody() method.

The body() of this class invokes call() on an coroutine of the inner anonymous coroutine. This anonymous coroutine is what invokes the attachedBody() of the subclass.

## 5.7 Status

This section discusses the status of the current implementation and suggests a few items for future work on the current code-base. As mentioned in section 5.5, exception handling is inherently missing from the current implementation.

Apart from exception handling, the implementation is considered complete in such a way that it can be used as a reference framework for flexible coroutines. However, no work has been put into making the framework efficient and light-weight. The code uses one thread per coroutine, and employs several semaphores (one per coroutine plus one per thread-call on a coroutine). Likewise, there has been no benchmarking of the code to compare it to other relevant models (see chapter 7) or to compare the different versions of the code whilst the code-base was evolving to its current incarnation.

#### Type safety of parameters and return values

The parameters and return values in the current implementation are both of type Object and there is no way for a coroutine to set any bounds on the number of parameters passed to it or their types. This is of course a weakness in the current implementation, type-safe coroutine would fit better into the Java language. Another weakness is that there is no way for a caller to determine if the coroutine returned null or if the coroutine returned no value. Besides being a weakness it leads to dangerous and inefficient code, the application code needs to use type-casts to obtain the correct types on parameters and return values.

Java 5.0 introduced *generics* as a way to declare parameterized types and methods. A flexible coroutine framework could employ generics to ensure type-safe calls, but this was not investigated any further.

#### Programmable scheduling policies

The current implementation has a hard-coded scheduling policy; coroutines are scheduled in a round-robin manner and they serve their callers FIFO. By hard-coded we mean that it is tightly coupled with the rest of the code and not a separate, programmable or configurable entity.

# Chapter 6

# Using flexible coroutines

This chapter presents examples that have been programmed using the flexible coroutine framework that was presented in the previous chapter. Section 6.1 solves the classical problem of *Readers and Writers* in different ways. Section 6.2 presents the colorful *Santa Claus Problem* and presents a solution and an alternative partial solution to it. Section 6.3 shows how binary and general semaphores can be implemented using flexible coroutines.

## 6.1 Readers and writers

The parallel programming literature is full of interesting and colorful problems that have been widely discussed and solved using a variety of synchronisation methods. Most of these problems can be easily understood but implementing them correctly can sometimes be challenging. One of these is the *Readers and Writers problem* which models access to a shared database. Besides being a great problem to compare and contrast synchronization mechanisms it is also an eminently practical problem.

In the Readers and Writers problem there are two kinds of processes that access the database. The *readers* execute transactions that only examine the database records. The *writers* will not only examine the records but also alter their contents. To guarantee a consistent database we say that a writer process must have exclusive access to the database. However, since the readers only read data, assuming there are, no writers accessing the database, any number of readers may concurrently execute their transactions. This definition that implies a shared database can of course be generalized, the processes can access a shared file, a shared in-memory data structure and so on.

This problem is a fine example of a what is called selective mutual exclusion. The different classes of processes, in this case the readers and the writers, compete for access to a shared database. However the readers and writers problem is also an example of a general condition synchronization problem since readers must wait until the condition *no writers are accessing the database* is true, and conversely the writers must wait until *no readers or no other writers are accessing the database* is true. This twofold view of the problem has led to a multitude of different solutions.

It is possible to imagine a number of possible policies when solving the readers and writers problem, the most straightforward solution is not always the most effective one if we imagine different sets of policies and preferences. Generally we can say that there are four policies:

- Serve the processes as they arrive, First-Come-First-Served (FCFS). No starvation as long as all processes release the lock.
- Give preference to *reader* processes. Can cause starvation if a continous stream of readers arrive while one or more writers are waiting.
- Give preference to *writer* processes. Can cause starvation if a continous stream of writers arrive while one or more readers are waiting.

Figure 6.1 shows a first attempt at a read-write lock using two flexible coroutines. The threads representing reader and writer processes have been omitted for brevity, but they can be seen in appendix A.2.

The RWLock class exports four methods, requestRead(), releaseRead(), requestWrite() and releaseWrite() that shows a common pattern in programming with flexible coroutines. These wrapper methods forward the request to private coroutine objects. Reader processes that want to gain access to the database simply call requestRead() on entry and releaseRead() when they are finished reading. As we see, this method invokes call() on the request coroutine with a parameter identifying the caller as a reader.

The private coroutines request and release is what actually performs the task of synchronizing the requests and releases on the lock. They do this by keeping a shared count of the number of readers and writers holding the lock. The fact that this read-write-lock is shared does not matter, remember that a flexible coroutine execute in mutual exclusion to other coroutines, meaning that only one coroutine will alter or read these shared variables at a time.

Let us take a closer look at how these two coroutines play together. First let us examine the request coroutine and see what happens if it is called by a reader named $r_1$. Remember that the thread of $r_1$ is blocked until request detaches:

- If there are no writers in the database, the readers count is increased and the request coroutine then detaches, letting another process attach to request if there are any.
- If there is a writer present, the request coroutine invokes passivate() , still attached to its caller $r_1$. Any future calls to request will now block and they will not be served until request has finished serving $r_1$.[1]

The situation is now that either we let the reader $r_1$ inside the database or it is blocked because there was a writer present. The idea is that when this

[1]They wait in the local queue of the request coroutine.

```
class RWFIFO {
    private static enum Parameter { READER, WRITER }
    private int readers, writers;
    private Request request = new Request("request");
    private Release release = new Release("release");

    public void requestRead() { Coroutine.call(request, Parameter.READER); }
    public void releaseRead() { Coroutine.call(release, Parameter.READER); }
    public void requestWrite() { Coroutine.call(request, Parameter.WRITER); }
    public void releaseWrite() { Coroutine.call(release, Parameter.WRITER); }

    private class Request extends SpinningCoroutine
        implements Parameterized {
        @Override
        public void spinningBody() {
            if (parameter0 == Parameter.READER) {
                if (writers > 0) {
                    passivate();
                } else {
                    ++readers;
                    detach();
                }
            } else if (parameter0 == Parameter.WRITER) {
                if (readers > 0 || writers > 0) {
                    passivate();
                } else {
                    ++writers;
                    detach();
                }
            } else {
                detach();
    } } }
    private class Release extends SpinningCoroutine
        implements Parameterized {
        @Override
        public void spinningBody() {
            if (parameter0 == Parameter.READER) {
                --readers;
            } else { // parameter0 == WRITER
                --writers;
            }
            if (readers + writers == 0) {
                kick(request);
            }
            detach();
    } }
}
```

**Figure 6.1:** Readers and writers.

writer releases the lock it invokes kick() on the request coroutine. If the request coroutine is passivated it is awakened and now it can check the number of writers again, this time assuring that it is actually zero and then detaching and releasing its caller. A similar set of events occur if the request coroutine is called by a writer.

We see that it is the asymmetric nature of call() and detach() combined with the ability to postpone the execution with passivate() that makes up the bulk of the read-write-lock.

This first attempt at a read-write-lock gives provides FCFS access to the shared resource and it allows multiple readers without affecting the FCFS principle. If there is a reader in the database and a writer request arrives, the request coroutine will passivate and keep attached to the writer. Thus any future calls on request block and wait in the local queue of the coroutine.

The question is then: could this solution be easily modified to give preference to reader or writer processes?

### 6.1.1 Prefer writer access

To give writers preference we need to ensure that:

- Incoming requests from readers are delayed if a writer is waiting.
- A delayed reader is awakened only if no writer is waiting.

Writers preference means that we want to let writers to access the database *before* readers, even if the readers asked for the lock before the writer. This means that a stream of writers can starve a reader.

The example above uses a single request object that application threads call() to request read and write access. The current implementation, as presented in chapter 5 and summarized in 5.7 imposes a strict scheduling policy: round-robin scheduling of the coroutines and threads served in FIFO-manner. This means that if we want writer preference and we have a writer $w$ accessing the database and a reader $r$ is attached to the request object, we want writer requests that arrive before $w$ has released the lock, to enter the database before $r$. However with the current implementation, these incoming requests are queued in the local queue of the request coroutine since it is attached to $r$. Figure 6.2 shows the local queue of the request coroutine with outstanding calls from both readers $(r_0, r_1)$ and writers $(w_0, w_1)$.

The example presented in figure 6.1 can not be easily modified to deal with such a scenario. To give writer processes access we would like for $w_0$ and $w_1$ to be scheduled before $r_0$ and $r_1$, but with the code as shown in figure 6.1 and the scheduling policy of the current implementation that is not possible.

A possible solution is to have different request coroutines for readers and writers, i.e. requestRead and requestWrite. That would result in two disjoint

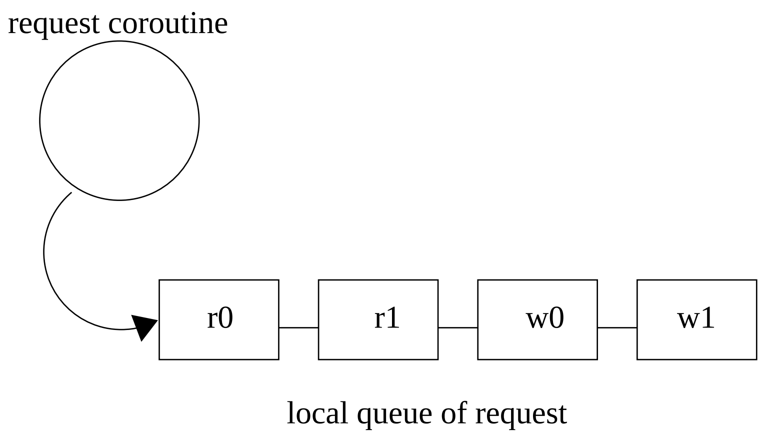

**Figure 6.2:** The local queue of the request coroutine.

queues for the two types of processes, and the scenario as seen in figure 6.1 is avoided by picking which queue to schedule from first.

Figure 6.3 shows an implementation of the read-write-lock that gives preference to writer processes. The wrapper methods have been omitted, the only change from figure 6.1 is that the two request methods for read and write, invoke call() on requestRead and requestWrite respectively.

The code is actually quite similar to the specification; we see that readers are waiting if there is a writer present or if a writer is waiting. The latter ensures that readers cannot go past a writer that is waiting for the lock. The requestRead coroutine is similar to the part in figure 6.1 where the caller was a writer process. The release coroutine has been altered slightly; it awakens both coroutines, i.e. it signals both queues.

### 6.1.2 Prefer reader access

The read-write-lock that gives writers preference upholds the policy by introducing yet another coroutine, in effect an extra queue. This extra queue lets us specify the kind of process we want to awaken.

We can modify the code in 6.3 to prefer reader access quite easily. In fact, it can be generalized even further, to be a reader *or* writer preference lock, configurable at run-time. This code is presented in appendix A.2.3. In this configurable version of the read-write-lock we employ a variable for checking which kind of process we prefer, and then kick *that* queue before the other. We also need to test whether there are any coroutines waiting in the local queue of the preferred process, before we dispatch the other kind.

```
class RWWriters {
    private static enum Parameter { READER, WRITER }
    private int readers, writers;
    private Coroutine requestRead = new RequestRead("requestRead");
    private Coroutine requestWrite = new RequestWrite("requestWrite");
    private Release release = new Release("release");

    private class RequestRead extends SpinningCoroutine {
        public void spinningBody() {
            if (writers > 0 || requestWrite.isWaiting()) {
                passivate();
            } else {
                ++readers;
                detach();
    }}}

    private class RequestWrite extends SpinningCoroutine {
        public void spinningBody() {
            if (readers > 0 || writers > 0) {
                passivate();
            } else {
                ++writers;
                detach();
    }}}

    private class Release extends SpinningCoroutine
        implements Parameterized {

        public void spinningBody() {
            if (p0 == READER) {
                --readers;
            } else { // p0 == WRITER
                --writers;
            }

            kick(requestWrite);
            kick(requestRead);

            detach();
    }}
}
```

**Figure 6.3:** Readers and writers with writers preference.

## 6.2 The Santa Claus problem

The *Santa Claus Problem* is an interesting and amusing exercise in concurrent programming. The problem is originally due to Trono [41] who solved it using classical semaphores. Benton [4] gave a brief summary of the problem:

> Santa repeatedly sleeps until wakened by either all of his nine reindeer, back from the holidays, or by a group of three of his ten elves. If awakened by the reindeer, he harnesses each of them to his sleigh, delivers toys with them and finally unharnesses them (allowing them to go off on holiday). If awakened by a group of elves, he shows each of the group into his study, consults them on toy R&D and finally shows them each out (allowing them to go back to work).
>
> Santa should give priority to the reindeer in the case that there is both a group of elves and a group of reindeer waiting.

The problem is a little more challenging than traditional mutual exclusion problems as it involves three sorts of processes that need to cooperate and synchronize at different places.

In this section we will show how the Santa Claus problem can be solved with flexible coroutines. Santa Claus, his reindeer and the elves will be represented by standard application threads; they will do their work in parallel. The flexible coroutines will be used to solve the difficult part of this problem, namely the synchronization of these three types of processes.

The complete solution can be found in A.3.

### 6.2.1 An auxiliary class

To help us implement the synchronization points we will use a multi-way rendezvous class [7], or a *n-way* as we will call it.

Threads can synchronize "around" a n-way object by means of the two public methods. One thread, that we will call the master, invokes accept() on the n-way. This causes the master thread to block until $n$ other threads have invoked entry(). Conversely, the calls to entry() will block until there has been a call to accept(). The $n$ that determines the number of threads required to release the master is given to the constructor.

The implementation of the n-way is split in two, first there is a binary rendezvous class, as seen in figure 6.4. This can be used to synchronize *two* threads. One thread calls entry(), the other thread calls accept(). The thread that arrives first to the rendezvous blocks until the other thread arrives.

This binary rendezvous uses two flexible coroutines and two wrapper methods that simply invoked Coroutine.call() on each of them. Implementing this with flexible coroutines proved itself to be very simple. They both check

```
public class Rendezvous {
    private Coroutine acceptCoroutine = new R();
    private Coroutine entryCoroutine = new R();
    public void accept() { Coroutine.call(acceptCoroutine); }
    public void entry() { Coroutine.call(entryCoroutine); }
    public Rendezvous() {
        accept.setOther(entry);
        entry.setOther(accept);
    }
    private class R extends SpinningCoroutine {
        private Coroutine other;
        public void setOther(Coroutine other) {
            this.other = other;
        }
        public void spinningBody() {
            if (other.caller() == null) { passivate(); }
            else { kick(other); }
            detach();
}}}
```

**Figure 6.4:** A binary rendezvous class.

the status of the other coroutine to determine which was called first, and then either passivates or kicks the other coroutine.

```
public class NWay {
    private Rendezvous rv = new Rendezvous();
    public void accept() {
        for (int i = 0; i < this.n; i++) {
            rv.accept();
    } }
    public void entry() { rv.entry(); }
}
```

**Figure 6.5:** A multi-way rendezvous class.

This binary rendezvous is then extended to a n-way rendezvous, as seen in figure 6.5. The NWay class is a simple wrapper around the Rendezvous class. The master thread invokes accept() on the binary rendezvous $n$ times and the $n$ other threads each invoke entry(). These invocations on entry() are queued in the flexible coroutine queue and they are therefore served in FIFO order with the current implementation.

## 6.2.2 A solution

The complete solution to the Santa Claus problem involves three different kinds of processes; ten elves, nine reindeer and Santa Claus himself. These processes will all be represented as application threads, so the only place we involve flexible coroutines is in the Rendezvous class.

However, the next section presents an alternative and better solution, where Santa Claus is implemented using a flexible coroutine, and not as a thread. Unfortunately the current implementation only supports one flexible coroutine system, and this solution demands two disjoint systems in order to execute correctly.

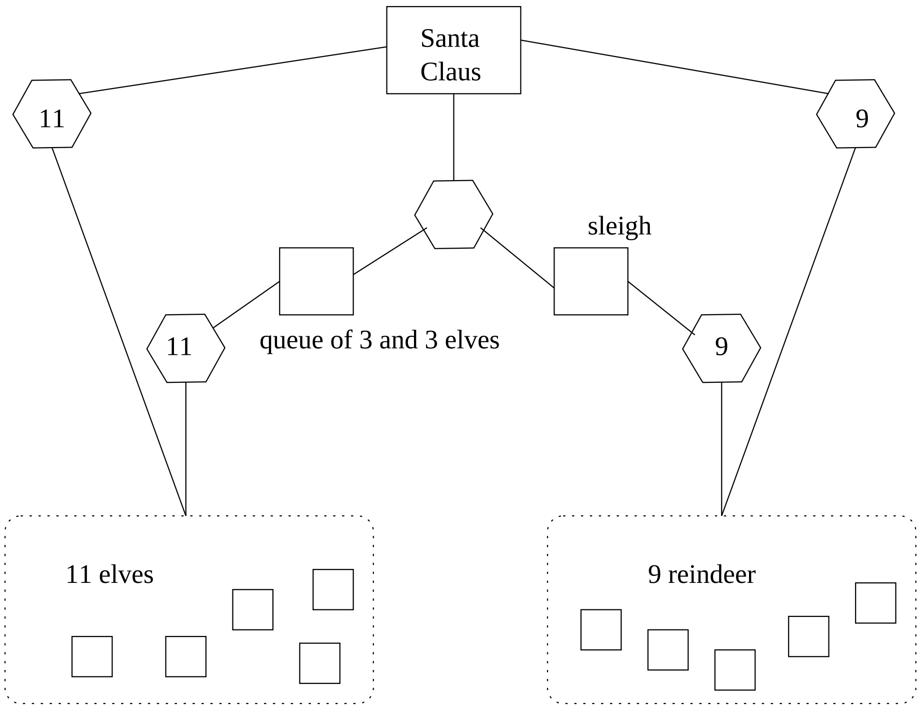

**Figure 6.6:** A solution to the Santa Claus problem.

Figure 6.6 shows an overview of the solution presented below. The polygons represent rendezvous points, the $n$ required to let the master through is depicted by the number inside the polygon. No number means this is a binary rendezvous.

Besides nine threads executing the Reindeer objects, ten threads executing the Elf objects and one thread executing Santa Claus, we have two synchronization points that also execute in their own threads: the reindeer meet in front of the sleigh when they have come back from vacation, and the last reindeer to come back also wakes up Santa by ringing his doorbell. The elves synchronize by forming queues of three and three, and whenever a queue of three is full they ring the doorbell.

When the reindeer come back from holiday they invoke entry() on the sleigh rendezvous point. On the other side of this rendezvous there is a sleigh that is waiting for each of the reindeer with accept(). When all reindeer are back they ring on Santa's doorbell through the binary doorbell rendezvous. Now, Santa takes them out for a ride and together they deliver toys. When they come back, the Santa thread invokes accept() on a rendezvous that every reindeer is waiting on, this represents the unharnessing of the reindeer. The interesting parts of the Reindeer class is shown in figure 6.7.

```
class Reindeer implements Runnable {
    public void run() {
        holiday(); // in Bahamas
        reindeerSleigh.enter(); // join all reindeer
        deliverToys(); // with Santa Claus
    }
    ⋮
    private void deliverToys() {
      // wait for Santa to unharness
      reindeerSantaSync.entry();
} }
```

**Figure 6.7:** The Reindeer class.

The same set of events occur when an elf discovers a problem. However, since elves only consult Santa in groups of three, they synchronize through an elf queue thread that is waiting on a 3-way rendezvous. This way, only three and three elves are let inside Santa's door. Elves are let out the door when Santa accept()'s a second 3-way rendezvous.

When the synchronization points (the queue of elves and the sleigh) ring on Santa's doorbell they also identify their mission by setting a special task variable in the Santa Claus object. Unfortunately, Santa is a plain old Java object and not a flexible coroutine. When these two synchronization point threads compete, i.e. they both want to ring Santa's doorbell, we have a race. It is imperative that Santa performs the same task as he was given. To eliminate this race, they need to synchronize the assignment of the task variable and the ringing of the doorbell with the Santa object, they do this in a synchronized-block. This can be seen in figure 6.8.

```
synchronized (santa) {
    santa.setTask(DELIVERTOYS);
    doorbell.entry();
}
```

**Figure 6.8:** Synchronizing on Santa Claus and ringing his doorbell.

### 6.2.3 An alternative solution

The solution sketched above and shown in appendix A.3 uses an application thread to execute the Santa Claus code. However, Santa Claus spends all his time waiting on the binary doorbell rendezvous.[2] This, coupled with the fact that he needs a variable telling him which task he is about to perform, begs the question: is Santa a flexible coroutine?

With Santa as a flexible coroutine, we could change the two synchronization threads so that they invoke call() with a parameter, instead of setting a task variable in the Santa object and then ringing the doorbell. With Santa as a flexible coroutine we could get rid of both the doorbell rendezvous and the synchronized blocks in the synchronization threads.

Figure 6.9 shows a simple spinning coroutine that could be used to implement Santa Claus. Notice that the coroutine checks its parameters to determine which task it is about to perform, i.e., which type of process invoked call on it. Once the task is finished, the coroutine passivates and wait for someone to call it again.

```
class Santa extends SpinningCoroutine implements Parameterized {
    public void spinningBody() {
        if (parameter0 == CONSULTELVES) {
            solveProblem();
            openDoor();
        } else { // p0 == DELIVERTOYS
            deliverToys();
            unharness(); }
        passivate();
} }
```

**Figure 6.9:** Santa Claus as a flexible coroutine.

With such an implementation of Santa Claus, the Reindeer and Elf threads can simply invoke call$(santa)$ instead of synchronizing on the Santa Claus object and then ringing the doorbell, as seen in figure 6.8.

There is a major problem with implementing Santa as a flexible coroutine. The Santa coroutine still needs to invoke accept() on the unharness n-way and on the n-way that lets elves out the door. Intuitively we would think that having a coroutine calling another coroutine (as would eventually happen, since the Rendezvous class is based on flexible coroutines) would not impose a problem. After all, coroutine-coroutine interaction is a part of the semantics. However the following paragraph will argue that the Santa Claus coroutine and the two coroutines compromising the Rendezvous class need to be in two, *disjoint* flexible coroutine *systems*. They are not only conceptually in different systems, they also need to operate in different systems in the current implementation.

The reason for this is the way we handle coroutines that use the AttachedOnly pattern. These check that they are attached at *dispatch-time*, i.e., they are scheduled and activated and *then* they check the condition. [3] If there

[2]Actually he is with Mrs. Claus in a warm and cozy bed. The implementation of the synchronization between Mrs. Claus and Santa is beyond the scope of this thesis.

[3]Whether or not the Rendezvous coroutines use this superclass does matter, they would still need to do this check themselves at dispatch-time.

had been a way for the coroutine system to check this condition *before* the coroutine is activated the problem would have been solved. However, since no such facility exist in the current implementation the Accept coroutine gets scheduled twice when called from a coroutine. A disjoint coroutine system would have solved this problem as well.

### 6.2.4 Summary

We have presented a solution to the Santa Claus problem. At the root of this solution we find multiple n-way synchronization points. These were built upon a binary rendezvous class that employed flexible coroutines.

The implementation of the binary rendezvous is relatively straightforward due to the sequential nature of the flexible coroutines. They each either wait or kick the other coroutine. The n-way that is wrapped around the binary rendezvous is also quite simple.

Unfortunately we found ourselves in the need for many of these rendezvous points and as we saw we could have gotten away with one less plus even simpler code if the flexible coroutine framework had supported multiple, disjoint systems.

The requirement that Santa gives preference to his reindeer was skipped, but it is believed that this could have been easily added if Santa Claus was implemented as a coroutine.

## 6.3 Semaphores

Section 3.3.2 discussed the coroutines as presented in Java 5.0. The implementation presented in chapter 5 relies heavliy on semaphores for synchronization and suspension of coroutine runner threads and application threads. In this section we show how semaphores can be implemented with flexible coroutines.

### 6.3.1 Binary semaphore

Remember that a binary semaphore is a semaphore whose only allowed values are zero (false) or one (true). We show an example of how one can implement a binary semaphore using flexible coroutines. For the sake of this example we will use only type of coroutine, but two instances of it. Figure 6.10 shows the code that we discuss in this section, the complete code is given in appendix A.4.

There are two wrapper methods up() and down() that each invoke call() on the corresponding coroutines, up and down that have been left out of this example, but that is found in the complete code. Notice that the coroutine extends the AttachedOnlyCoroutine special coroutine, so that its body only executes if it is attached to a process.

```
public class BinarySemaphore {
    private Coroutine up, down;
    private boolean value;
    public BinarySemaphore() { this(true); }
    public BinarySemaphore(boolean initial) {
        value = initial;
        up = new BinSem(true);
        down = new BinSem(false);
        up.other = down;
        down.other = up;
    }
    private class BinSem extends AttachedOnly {
        private boolean pval;
        public Coroutine other;
        public BinSem(boolean pval) { this.pval = pval; }
        public void attachedBody() {
            while (true) {
                if (pval) {
                    passivate();
                } else {
                    value = pval;
                    kick(other);
                    detach();
                }
            }
    }}}
```

**Figure 6.10:** Binary semaphore.

Most of the complexity of this code comes from the fact that we only use one coroutine, it would have been substantially easier to understand if we had used two coroutines, i.e. one for up() and for down().

We can see how the code works if we first define the two operations using the folloing pseudo-code:

```
down:   await (v == true) then v = false
  up:   await (v == false) then v = true
```

The code in the inner BinSem coroutine is then easier to comprehend; each of the two coroutines are initialized with trueand falseand then act accordingly. If we substitute the variable pval with truein the upcoroutine and falsein the downcoroutine we see that the code is in fact quite the same as the pseudo-code given here.

### 6.3.2 General semaphore

A general semaphore is one that can take any non-negative value. We can define the two operations with pseudo-code as:

```
down:   await (s > 0) then s = s - 1
  up:   s = s + 1
```

We see that the up-operation will always increment the semaphore. The down-operation however, need to block until the semaphore is greater than zero before it can decrement. Both operations need to execute atomically.

The code in figure 6.11 implements a general semaphore using two internal coroutines. We see that the coroutines Up and Down follow from the definition. The await part is substituted with a passivate() until the condition is true. The Up coroutine kicks the Down coroutine when the value changes.

```
public class GeneralSemaphore {
    private int value;
    public GeneralSemaphore(int initial) {
        value = initial;
        down = new Down();
        up = new Up(down);
    }
    private class Up extends SpinningCoroutine {
        public Coroutine down;
        public Up(Coroutine down) { this.down = down; }
        public void spinningBody() {
            ++value;
            kick(down);
    }}
    private class Down extends AttachedOnly {
        public void attachedBody() {
            while (true) {
                while (value < 1) passivate();
                --value;
                detach();
    }}}
}
```

**Figure 6.11:** General semaphore.

# Chapter 7

# Related work

This chapter presents related work. Sequential Object Monitors in Java in section 7.1 and Polyphonic C♯ in section 7.2, an extension to the C♯ programming language.

## 7.1 Sequential Object Monitors

Sequential object monitors [6] is a recent proposal meant as an alternative to programming with standard Java object monitors. Programming with Java object monitors is widely recognized to be difficult and error-prone and in many cases also inefficient due to the many context switches introduced by the notifyAll() primitive. Sequential object monitors (SOM) is introduced as a new concurrency abstraction and a library providing the necessary tools to work with them in Java is presented along with a few examples of usage.

### 7.1.1 Overview

The authors mention several motivation points that have influenced their work on SOMs; easy to use, powerful enough to express any concurrency abstraction, efficient and modular. The most important point is perhaps the last keyword, that it is modular. SOMs separate the synchronization code from the application code making it easy to "plug-in" existing synchronization code onto code that is not thread-safe. In addition to all that they also aim to be portable, their solution is written in 100% Java, there is no need for a special virtual machine.

A sequential object monitor is a plain old java object (POJO) to which a low-cost, thread-less *scheduler* object is attached, like shown in figure 7.1. Notice that the scheduler is in fact thread-less, the SOM does not have its own thread of control. The functional code, i.e. the application logic, is in a standard Java object without any synchronization code. The synchronization code is separated from the application logic and localized in the scheduler object. This

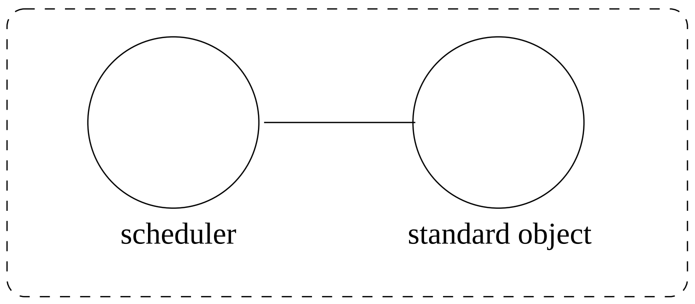

**Figure 7.1:** A sequential object monitor.

object implements a scheduling method which is responsible for specifying how concurrent requests to the application object should be scheduled. This makes it possible to attach different schedulers to objects of the same class at run-time. The methods exported by the monitor are those that are public in the standard Java object.

When a call comes in from a thread to the monitor, that is to say when a thread calls one of the methods in the POJO, it is automatically turned into a request object and then queued in a pending queue until it is scheduled by the scheduler object. The scheduler then marks request for execution. The requests are then safely executed in mutual exclusion to the other scheduled requests.

The SOM is *sequential* in the sense that thread interleaving is not necessary when writing the functional code, the method body is always executed atomically from beginning to end. Compared to Java monitors, where several invocations on a synchronized method can co-exist (although only one can be active at a time), the sequential nature of the SOMs makes it easier to reason about the program.

### 7.1.2 Implementation

One of the main goals for SOMs was that it should be portable and that it should not require a modified virtual machine. In order to transparently reify method calls to the functional object and turn them into request objects, the SOM library is based on a reflective infrastructure that is operating at load time. The SOM meta-object protocol is defined within a behavioral reflective extension of Java called Reflex [40].

When creating a SOM the reflective infrastructure ensures that method invocations on the function object are intercepted and a meta-object controller is invoked. This controller is invoked before the method is requested and just after the method call completes. It is this SOM meta-object that ensures that the scheduling is invoked and that the requests are scheduled in mutual exclusion of concurrent requests.

The library presented is a complex and efficient piece of machinery. Several benchmarks are presented where code employing SOMs outrun code with standard Java monitors. The complexity of the implementation however does not shine through and developing applications with SOMs is meant to be easy and straight-forward.

### 7.1.3 Compared to flexible coroutines

SOMs provide features that can be compared to those of flexible coroutines. The feature that SOMs provide that lack in the current implementation of flexible coroutines is that of a programmable and configurable scheduling semantics. Note that this is a lacking feature in the current implementation, not in the formal semantics. See the future work section in chapter 8.

## 7.2 Polyphonic C♯

C♯ is an object-oriented programming language [16] developed by Microsoft as a part of their .NET-initiative. Its syntax is based on C, C++ and Java and just like Java it is intended to compile into code that is executed on a virtual machine [17]. Not only does C♯ and Java share similar syntax; their thread models are also very much alike.

Polyphonic C♯ is an extension to C♯ with new asynchronous concurrency constructs, based on join calculus [3].

### 7.2.1 Overview

Polyphonic C♯ adds two new concepts to the conventional object-oriented programming model of C♯; *asynchronous method* and *chords*.

Conventional methods and functions are synchronous in the sense that the caller is blocked until the callee completes. In Polyphonic C♯ it is possible to mark a method as asynchronous. Any call to such a method is guaranteed to return immediately. As such, asynchronous can never return a result and calling an asynchronous method is very much like posting an event or sending a message. An asynchronous method is declared by using the async keyword instead of void:

```
async postEvent(EventInfo data) {
    // method body
}
```

When a thread invokes an asynchronous method the call returns immediately. However, the method body is scheduled to execute in a different thread, either a new one that spawns to serve this method or an existing from a threadpool. However, this kind of usage is rare in Polyphonic C♯. The more common approach is to use asynchronous methods together with *chords* without necessarily demanding a new thread.

A *chord* consists of a header and a body. The *header* is a set of method declarations and the *body* is only executed one *all* the methods in the header have been called. A short example:

```
class Buffer {
    string Get() & async Put(string s) {
        return s;
    }
}
```

This code defines a class with two instance methods that belong together in a single chord. On an instance of this class, each call to Get() is *matched* with a call to the asynchronous method Put(). Outstanding Put() calls are queued until a matching Get() arrives, and a Get() call that has no matching Put() call is blocked until another thread supplies a matching Put().

# Chapter 8

# Conclusion

Captain's journal. Stardate April 13. We have failed to uphold Brannigan's Law. However, I did make it with a hot alien babe.

Zap Brannigan, Futurama

This chapter recapitulates the main results of this thesis. Section 8.1 summarizes the contributions this thesis has made and 8.2 mentions possible points of improvements. Section 8.3 explores opportunities for future work.

## 8.1 Contribution

The contributions are listed below in order of importance and then discussed further. The contributions address the issues raised in the introduction of this thesis.

- The introduction of a formal semantics for the scheduling of flexible coroutines. This made it possible to include parameters and return values into the formal semantics.
- The creation of a reference framework for programming with flexible coroutines in Java that implements the formal semantics including parameters and return values.
- Example usage of flexible coroutines, showing that flexible coroutines are in fact usable and that they can solve a variety of problems.
- A brief discussion of how flexible coroutines relate to other recent proposals that aim to simplify the programming of parallel systems.

The basic formal semantics as presented by Belsnes and Østvold did not include scheduling nor parameters and return values. This thesis has introduced a formal semantics for scheduling by extending the basic semantics. The scheduling rule coupled with the two attachment rules, allowed for simple inclusion of parameters and return values into the semantics. This extension has led to simpler implementation and a broader understanding of flexible coroutine systems. By identifying problems and undefined scenarios in the original semantics, this has led to more a robust semantics.

The extended semantics simplified the implementation of parameters and return values as it clearly defines what configurations and coroutine states lead to a new set of parameters being passed. That parameters should not be passed at dispatchment time did not match the authors intuition. This is why early versions of the code passed parameters at dispatch time in the concrete scheduler.

The creation of a framework for programming with flexible coroutines in Java has made it possible to experiment with real programs and problems. This implementation has shown itself to be extensible and usable. The code is closely related to the semantics, and can therefore be read side by side with the formal semantics to gain a deeper understanding.

The thesis can also be considered a discussion of how well threads can be a part of a language that incorporates the "all objects are coroutines" concept, as found in Simula. In such a language, semantics for interaction of objects that behave like coroutines and threads needs to defined.

## 8.2 Critique

This thesis has presented an extension of the basic formal semantics for flexible coroutines, which hopefully will prove to be a valuable contribution to programmers and theorists that wish to further explore the relationship of flexible coroutines and related proposals.

For a long period of time the semantics was reasonably complex and quite hard to understand. This substantially slowed down the effort of creating real-life examples that employed flexible coroutines. The scheduling semantics as presented in this thesis is simple and general. However, the actual order of scheduling coroutines is left outside the semantics and decided by a policy. No work has been put into experimenting with different scheduling policies. By experimenting with different scheduling policies it is possible that we could have generalized the semantics further, or even noticed flaws in the scheduling semantics. It is also possible that this could have lead to an understanding of flexible coroutine systems in such a way that would let us simplify the semantics.

The semantics as presented here is reasonably clear, but because of to the nature of Simula coroutines, i.e. both symmetric *and* asymmetric, the semantics for flexible coroutine has a few dark corners. The exact meaning of thread-interaction on coroutines with callee chains could have been investigated more closely.

The implementation presented is considered complete. However, as mentioned in section 5.5, exception handling is missing. Section 5.7 mentions other missing parts of the implementation. It *is* possible to add some form of exception handling to the current implementation, but this was not considered an important feature.

The current implementation uses a hard-coded scheduling policy. The lack of a programmable or configurable scheduling policy means that it is not possible to experiment with any other type of scheduling. Having this ability would have lead to an implementation that is closer to Sequential Object Monitors and it would have made it substantially easier to program some of the examples. The examples does not leverage the coroutine system to the full, symmetric coroutines via the resume operation are never used, and it would have been interesting to see a larger example that includes a large-scale coroutine system with both asymmetric and symmetric coroutines that interact with threads.

Even though a brief comparison with Sequential Object Monitors were given, the relationship between these and flexible coroutines could definitely be investigated more closer. An interesting exercise would be to implement sequential object monitors using flexible coroutines, and vice versa. Unfortunately, lack of time prevented us from doing so.

## 8.3 Future work

Instead of extending the formal semantics for flexible coroutines it would be interesting to take a step back, and then in the spirit of Belsnes and Østvold start with extending the formal semantics for Simula coroutines with parameters and return values. This could perhaps be generalized into the semantics for flexible coroutines, and hopefully lead to simpler semantics. Besides extending Simula coroutines with parameters and return values, it should be possible to let Simula coroutines be able to call other coroutines that are attached, and then block until the callee coroutine detaches.

Also, since the semantics is still considered reasonably complex, it could be useful to provide detailed guidelines for programming with flexible coroutines. Many configurations can lead to systems that block, but many of these transitions can be avoided if the programmer follows a few simple rules.

Identifying the parts of the code that deal with the scheduler and then reformulate this so that it allows for programmable or configurable scheduling is considered an important improvement of the current code. Likewise, if we want to employ flexible coroutines in larger systems, with multiple coroutine systems, we should consider implementing type-safe parameter passing and return values, plus better exception handling.

An interesting part that was left out of this thesis was benchmarking the flexible coroutine framework. This was never considered as important as getting a correct implementation, nevertheless it would have been interesting to see the efficiency of it compared to, for example, Sequential Object Monitors. This could be done side-by-side with a more theoretical comparison of the expressiveness of SOMs and flexible coroutines.

# Appendix A

# Code listings

## A.1 Flexible coroutines in Java

### A.1.1 Coroutine.java

```
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

/**
 * author: Steingrim Dovland
 * version: 1.0
 */

interface Parameterized {
    void setUpParameters(Object[] params);
}

interface Returning {
    Object getReturnValue();
}

public abstract class Coroutine {

    private enum State {
        ACTIVE, CALLING, IDLE, TERMINATED, WAITING, ERROR
    }

    private enum OperationType {
        CALL, DETACH, KICK, PASSIVATE, RESUME, YIELD
    }

    public Coroutine() {
        this.state = State.IDLE;
    }

    public Coroutine(String name) {
        this.name = name;
        this.state = State.IDLE;
    }
```

```
abstract void body();

/*
 * coroutine operations
 */
public final static Object call(Coroutine next) {
    return call(next, new Object[0]);
}

public final static Object call(Coroutine next, Object... params) {
    // this method can be invoked from both Coroutine instance
    // threads and regular threads. this is the only method where it
    // actually matters whether it is a Runner or a regular thread.

    Operation op;
    Object retval;

    systemLock.lock();

    //
    // coroutine-coroutine call
    //
    if (Thread.currentThread() instanceof Coroutine.Runner) {

        // we're executing this method in a Runner thread which
        // means that the caller is systemCurrent.

        op = new Operation(OperationType.CALL, systemCurrent);

        next.operation = op;
        next.caller = systemCurrent;
        if (next instanceof Parameterized) {
            ((Parameterized)next).setUpParameters(params);
        }

        while (next.callee != null)
            next = next.callee;

        if (!systemQueue.contains(next))
            systemQueue.add(next);

        // we won't wake up until 'next' detaches so that is when
        // we'll fetch those pesky return values.
        schedule(true);

        // and we're awake again. get that return value.
        if (next instanceof Returning) {
            retval = op.getReturnValue();
        } else {
            retval = null;
        }

    //
    // thread-coroutine call
    //
    } else {

        op = new Operation(OperationType.CALL, Thread.currentThread());
        boolean sched = false;

        if (next.operation != null) {
                next.localQueue.add(op);
                if (next instanceof Parameterized) {
                    op.setParameters(params);
                }
            }
            else {
                next.operation = op;
                sched = true;
                if (next instanceof Parameterized) {
                    ((Parameterized)next).setUpParameters(params);
                }
            }

            while (next.callee != null)
                next = next.callee;

            if (!systemQueue.contains(next))
                systemQueue.add(next);

            if (sched)
                schedule(false);

            // we're executing this method in a regular thread. there
            // *may* be a coroutine currently executing, therefore go
            // with non-forced scheduling.

            systemLock.unlock();

            try {
                // block the calling thread (i.e. this thread)
                op.threadBlockingSemaphore.acquire();
                // ok. we're awake. time to get those return values.

                if (next instanceof Returning) {
                    retval = op.getReturnValue();
                } else {
                    retval = null;
                }

            } catch (InterruptedException ie) {
                throw new RuntimeException("interrupted while sleeping: " +
                                            ie.getMessage());
            }
        }

        return retval;
    }

    public final static void kick(Coroutine next) {
        // this method can be invoked from both Coroutine instance
        // threads and regular threads

        systemLock.lock();

        while (next.callee != null)
            next = next.callee;

        if (next.isActive() || systemQueue.contains(next))
            next.localQueue.add(new Operation(OperationType.KICK));

        if (!systemQueue.contains(next))
            systemQueue.add(next);
        // if we're executing in a runner thread then systemCurrent will
        // be non-null which means this schedule() call will be a no-op.
        // if we're in a regular thread then 'next' will only be invoked
        // if systemCurrent is non-null.
        schedule(false);
        systemLock.unlock();
    }

    protected final static void detach() {
        detach(State.IDLE);
    }

    protected final static void detach(State newState) {
        // this method can only be invoked from a Coroutine instance
        // threads, not from regular threads

        systemLock.lock();

        // it will *always* be systemCurrent who is detaching. anything
        // else is just plain wrong.
        systemCurrent.state = newState;

        // if caller is a coroutine then we need to add that to the
        // systemQueue. if caller is a thread we need to release it. if
        // there is no caller, just go idle.

        if (systemCurrent instanceof Returning && systemCurrent.operation != null) {

            systemCurrent.operation.setReturnValue(
                ((Returning)systemCurrent).getReturnValue());
        }

        if (systemCurrent.operation != null &&
            systemCurrent.operation.isCoroutineCall()) {
            systemQueue.add(systemCurrent.caller);

            // and wipe out the relationship
            systemCurrent.caller.callee = null;
            systemCurrent.caller = null;

        } else if (systemCurrent.operation != null &&
                   systemCurrent.operation.isThreadCall()) {

            // this will release the thread
            systemCurrent.operation.threadBlockingSemaphore.release();
        } // else; we have no caller so don't do anything special

        systemCurrent.operation = null;

        // if there are still operations pending on this detaching
        // coroutine, make sure that they're added to systemQueue.
        // (i'm pretty sure that the second clause is redundant)
        if (!systemCurrent.localQueue.isEmpty()) {
            Operation op = systemCurrent.localQueue.poll();
            if (systemCurrent instanceof Parameterized) {
                ((Parameterized)systemCurrent).setUpParameters(op.getParameters());
            }

            systemCurrent.operation = op;
            if (op.isCoroutineCall())
                systemCurrent.callee = op.coroutineCaller;
            if (!systemQueue.contains(systemCurrent))
                systemQueue.add(systemCurrent);
        }

        schedule(true);
    }

    protected final static void passivate() {
        // this method can only be invoked from a Coroutine instance
        // threads, not from regular threads

        systemLock.lock();
        systemCurrent.state = State.IDLE;

        // the lock will be released deep down in enter() or in leave()
        schedule(true);
    }

    protected final static void yield() {
        // this method can only be invoked from Coroutine instance
        // threads, not from regular threads

        systemLock.lock();
        systemCurrent.state = State.WAITING;
        systemQueue.add(systemCurrent);

        // the lock will be released deep down in enter()
        schedule(true);
    }

    protected final static void resume(Coroutine next) {
        // this method can only be invoked from Coroutine instance
        // threads, not from regular threads

        systemLock.lock();
        while (next.callee != null) next = next.callee;
        if (!systemQueue.contains(next))
            systemQueue.add(next);

        // the lock will be released deep down in enter()
        schedule(true);
    }

    /*
     * coroutine system
     */

    private static Coroutine systemCurrent = null;

    private static Queue<Coroutine> systemQueue = new LinkedList<Coroutine>();

    private static Lock systemLock = new ReentrantLock();

    private static void schedule(boolean forced) {
        // the thread executing this method will *always* own systemLock

        // this means we've been called but that we should not do
        // anything if there is a coroutine executing
        if (!forced && systemCurrent != null) {
            // get out of here and let caller release systemLock
            return;
        }

        // if the queue is empty then the whole system should be put to
        // sleep, but remember we own the systemLock!
        if (systemQueue.isEmpty()) {
            Coroutine coroutine = systemCurrent;
            systemCurrent = null;
            // leave() will release systemLock
            coroutine.leave();
            // and when we're awakened we just return
            return;
        }

        Coroutine next = systemQueue.poll();

        next.state = State.ACTIVE;
        next.enter();
    }

    /*
     * coroutine instances
     */

    private Coroutine callee;

    private Coroutine caller;

    private Operation operation;

    private String name;

    private Runner runner;

    private State state;

    private Queue<Operation> localQueue = new LinkedList<Operation>();

    private Semaphore semaphore = new Semaphore(0);

    private void enter() {
        // the thread executing this method will *always* own systemLock

        if (runner == null) runner = new Runner(this);

        // "enter" the coroutine, i.e. start executing where it last
        // left off or at the beginning. there are two scenarios if we
        // take a look up the stack
        //
        // 1) coroutine-coroutine interaction
        // 2) thread-coroutine interaction
        //
        // if 1) then we are executing in a Runner and if 2) we are
        // executing in the thread of our caller/kicker.

        // this only fits in 2) and so we're either kicked or called –
        // go() will start the runner and we should return to caller
        // which will release systemLock.
        if (systemCurrent == null) {
            systemCurrent = this;
            systemCurrent.runner.go();
            return;
        }
        // ok, we now know that there *is* a systemCurrent and by very
        // good reasons that means that we're in a Runner.

        // switch systemCurrent with this and keep a reference to the
        // coroutine which thread we execute in. this is safe because we
        // own the systemLock, remember.
        Coroutine previous = systemCurrent;
        systemCurrent = this;

        // start the new current and let go of the systemLock
        systemCurrent.runner.go();
        systemLock.unlock();

        // remember we're still in the thread of the previous
        // systemCurrent. if this is terminated it means it has fallen
        // of the end of its body(). just return and let this thread
        // die a silent death.
        if (previous.isTerminated()) {
            return;
        }

        // mkay, this coroutine is not done yet but it needs to be put
        // to sleep, or else it'll fall back to its body() eventually.
        // (systemLock has been released)
        try {
            previous.semaphore.acquire();
        } catch (InterruptedException ie) {
            throw new RuntimeException("interrupted while sleeping: " +
                                       ie.getMessage());
        }
    }

    private void leave() {
        // the thread executing this method will *always* own systemLock

        // if we were co-called then we won't get here, however if we
        // were thread-called then we may get here since the last thread
        // call will leave the systemQueue as empty. if so we need to
        // awaken our caller!
        systemCurrent = null;
        systemLock.unlock();

        state = State.IDLE;

        try {
            semaphore.acquire();
        } catch (InterruptedException ie) {
            throw new RuntimeException("interrupted while sleeping: " +
                                       ie.getMessage());
        }
    }

    private class Runner extends Thread {

        private Coroutine coroutine;

        public Runner(Coroutine coroutine) {
            this.coroutine = coroutine;
            setName(coroutine.getName() + "_runner");
        }
        public void go() {
            // this method will be executed by another thread that holds
            // systemLock, but never by *this* thread. simply
            // start/release the thread, if it tries to do anything bad
            // it will need to get the systemLock which will fail.
            if (!isAlive()) {
                start();
            } else {
                coroutine.semaphore.release();
            }

            // our caller will release systemLock
        }

        public void run() {
            coroutine.body();
            coroutine.state = Coroutine.State.TERMINATED;
            detach();
        }
    }

    private static class Operation {
        OperationType operation;
        Thread threadCaller;
        Coroutine coroutineCaller;
        Semaphore threadBlockingSemaphore;
        Object[] parameters;
        Object returnValue;

        Operation(OperationType optype) {
            operation = optype;
        }

        Operation(OperationType optype, Thread caller) {
            assert optype == OperationType.CALL;

            operation = optype;
            threadCaller = caller;
            threadBlockingSemaphore = new Semaphore(0);
        }

        Operation(OperationType optype, Coroutine caller) {
            assert optype == OperationType.CALL;

            operation = optype;
            coroutineCaller = caller;
        }

        void setParameters(Object[] params) {
            assert operation == OperationType.CALL;
            parameters = params;
        }

        Object[] getParameters() {
            assert operation == OperationType.CALL;
            return parameters;
        }

        void setReturnValue(Object value) {
            assert operation == OperationType.CALL;
            returnValue = value;
        }
        Object getReturnValue() {
            assert operation == OperationType.CALL;
            return returnValue;
        }

        boolean isCoroutineCall() {
            return operation == OperationType.CALL &&
                coroutineCaller != null;
        }

        boolean isThreadCall() {
            return operation == OperationType.CALL &&
                threadCaller != null;
        }
    }

    /*
     * misc public helper methods
     */
    public boolean isTerminated() {
        return state == State.TERMINATED;
    }

    public boolean isActive() {
        return state == State.ACTIVE;
    }

    public boolean isWaiting() {
        return state == State.WAITING;
    }

    public boolean isIdle() {
        return state == State.IDLE;
    }

    public Object caller() {
        if (operation == null) {
            return null;
        }

        if (operation.isCoroutineCall()) {
            return operation.coroutineCaller;
        } else if (operation.isThreadCall()) {
            return operation.threadCaller;
        } else {
            return null;
        }
    }

    public void setName(String name) {
        this.name = name;
        if (this.runner == null) this.runner = new Runner(this);
        this.runner.setName(name + "_runner");
    }

    public String getName() {
        return name;
    }

    public String toString() {
        return name + "[" + state + "]";
    }
}
```

### A.1.2 SpinningCoroutine.java

```
abstract class SpinningCoroutine extends Coroutine {

    public SpinningCoroutine() {
        super();
    }

    public SpinningCoroutine(String name) {
        super(name);
    }

    @Override
    void body() {
        while (true) {
            spinningBody();
        }
    }

    abstract void spinningBody();
}
```

### A.1.3 AttachedOnlyCoroutine.java

```
abstract class AttachedOnlyCoroutine extends Coroutine {
    Coroutine c = new Coroutine() {
        public void body() { attachedBody(); }
    };

    void body() {
        while (true) {
            if (this.caller() != null)
                call(c);
            detach();
        }
    }

    abstract void attachedBody();
}
```

## A.2 Readers and writers

### A.2.1 Reader and writer threads

```
import org.apache.log4j.Logger;

class RWFIFO {
```

```
static Logger LOGGER = Logger.getLogger(RWFIFO.class);

static RWFIFOLock rwlock = new RWFIFOLock();

public static void main(String[] args) {
    final int nreaders = 3;
    final int nwriters = 1;

    Thread.currentThread().setName("main");

    for (int i=0; i < nreaders; ++i) {
        Thread t = new Reader(i);
        t.setName("reader_thread_"+i);
        t.start();
    }

    for (int i=0; i < nwriters; ++i) {
        Thread t = new Writer(i);
        t.setName("writer_thread_"+i);
        t.start();
    }

    LOGGER.info("main is done");
}

private static class Reader extends Thread {
    private int name;

    Reader(int name) {
        this.name = name;
    }

    public void run() {
        final int ntimes = 1;

        for (int i = 0; i < ntimes; ++i) {
            TestUtils.sleepRandom();

            LOGGER.info("Reader " + name + " wants to read");
            rwlock.requestRead();

            LOGGER.info("Reader " + name + " is reading");
            TestUtils.sleepRandom();

            rwlock.releaseRead();
            LOGGER.info("Reader " + name + " stopped reading");
        }

        LOGGER.info("Reader " + name + " completed");
    }

    public String toString() {
        return "Reader_" + this.name;
    }
}

public static class Writer extends Thread {
    private int name;

    Writer(int name) {
        this.name = name;
        }

        public void run() {
            final int ntimes = 2;

            for (int i=0; i < ntimes; ++i) {
                TestUtils.sleepRandom();
                LOGGER.info("Writer " + name + " wants to write");
                rwlock.requestWrite();

                LOGGER.info("Writer " + name + " is writing");
                TestUtils.sleepRandom();

                rwlock.releaseWrite();
                LOGGER.info("Writer " + name + " stopped writing");
            }

            LOGGER.info("Writer " + name + " completed");
        }

        public String toString() {
            return "Writer_" + this.name;
        }
    }
}
```

### A.2.2 RWFIFO.java.tex

```
class RWFIFO {
    private static enum Parameter { READER, WRITER }
    private int readers, writers;
    private Request request = new Request("request");
    private Release release = new Release("release");

    public void requestRead() { Coroutine.call(request, Parameter.READER); }
    public void releaseRead() { Coroutine.call(release, Parameter.READER); }
    public void requestWrite() { Coroutine.call(request, Parameter.WRITER); }
    public void releaseWrite() { Coroutine.call(release, Parameter.WRITER); }

    private class Request extends SpinningCoroutine
        implements Parameterized {
        @Override
        public void spinningBody() {
            if (parameter0 == Parameter.READER) {
                if (writers > 0) {
                    passivate();
                } else {
                    ++readers;
                    detach();
                }
            } else if (parameter0 == Parameter.WRITER) {
                if (readers > 0 || writers > 0) {
                    passivate();
                } else {
                    ++writers;
                    detach();
                }
            } else {
                detach();
    } } }
    private class Release extends SpinningCoroutine
        implements Parameterized {
        @Override
        public void spinningBody() {
            if (parameter0 == Parameter.READER) {
                −−readers;
            } else { // parameter0 == WRITER
                −−writers;
            }
            if (readers + writers == 0) {
                kick(request);
            }
            detach();
    } }
}
```

### A.2.3 RWPreference.java

```
class RWPreference {
    private static enum Parameter { READER, WRITER }
    private int readers, writers;
    private boolean writersPref;
    private Coroutine requestRead = new RequestRead("requestRead");
    private Coroutine requestWrite = new RequestWrite("requestWrite");
    private Release release = new Release("release");

    private class RequestRead extends SpinningCoroutine {
        public void spinningBody() {
            if (writers > 0 ||
                (writersPref && requestWrite.isWaiting())) {
                passivate();
            } else {
                ++readers;
                detach();
            }
        }
    }

    private class RequestWrite extends SpinningCoroutine {
        public void spinningBody() {
            if (readers > 0 || writers > 0 ||
                (!writersPref && requestRead.isWaiting())) {
                passivate();
            } else {
                ++writers;
                detach();
            }
        }
    }

    private class Release extends SpinningCoroutine
        implements Parameterized {
        public void spinningBody() {
            if (p0 == READER) {
                −−readers;
            } else { // p0 == WRITER
                −−writers;
            }

            if (writersPref) {
                kick(requestWrite);
                kick(requestRead);
            } else {
                kick(requestRead);
                kick(requestWrite);
            }

            detach();
        }
    }
}
```

## A.3 The Santa Claus problem

### A.3.1 Rendezvous.java

```
public class Rendezvous {
    private Coroutine acceptCoroutine = new R();
    private Coroutine entryCoroutine = new R();

    // up
    public void accept() { Coroutine.call(acceptCoroutine); }
    // down
    public void entry() { Coroutine.call(entryCoroutine); }

    public Rendezvous() {
        accept.setOther(entry);
        entry.setOther(accept);
    }

    private class R extends SpinningCoroutine {
        private Coroutine other;
        public void setOther(Coroutine other) {
            this.other = other;
        }
        public void spinningBody() {
            if (other.caller() == null) {
                passivate();
            } else {
                kick(other);
            }
            detach();
        }
    }
}
```

### A.3.2 NWay.java

```
public class NWay {
    private int n;
    private Rendezvous rv = new Rendezvous();
    public NWay(int n) {
        this.n = n;
    }

    public void accept() {
        for (int i = 0; i < n; i++) {
            rv.accept();
        }
    }

    public void entry() {
        rv.entry();
    }
}
```

### A.3.3 SantaClaus.java

```
public class SantaClaus {
    final static int nReindeer = 9;
    final static int nElves = 11;

    final static int DELIVERTOYS = 1;
    final static int CONSULTELVES = 2;

    static Santa santa = new Santa();
    static Sleigh sleigh = new Sleigh();
    static ElfQueue elfqueue = new ElfQueue();

    static NWay reindeerSleighSync = new NWay(nReindeer);
    static NWay reindeerSantaSync = new NWay(nReindeer);
    static NWay elvesSantaSync = new NWay(3);
    static NWay elvesQueueSync = new NWay(3);
    static Rendezvous doorbell = new Rendezvous();

    public static void main(String[] args) {
        System.out.println("creating threads...");

        new Thread(santa).start();
        new Thread(sleigh).start();
        new Thread(elfqueue).start();

        for (int i=0; i < nReindeer; i++)
            new Thread(new Reindeer(i)).start();

        for (int i=0; i < nElves; i++)
            new Thread(new Elf(i)).start();
    }

    static class Santa implements Runnable {
        private int task;

        public void run() {
            Thread.currentThread().setName("santathread");

            while (true) {
                System.out.println("santa going to sleep");
                doorbell.accept();
                System.out.println("santa awakened");
            if (task == DELIVERTOYS) {
                deliverToys();
                unharness();
            } else {
                solveProblem();
                openDoor();
            }
        }
    }

    private void deliverToys() {
        System.out.println("santa delivering toys");
        sleep(2, 6);
    }

    private void solveProblem() {
        System.out.println("santa solving problems");
        sleep(2, 6);
    }

    private void unharness() {
        System.out.println("santa unharnessing");
        reindeerSantaSync.accept();
        System.out.println("santa done unharnessing");
    }

    private void openDoor() {
        System.out.println("santa letting out elves");
        elvesSantaSync.accept();
    }

    public void setTask(int task) {
        this.task = task;
    }
}

static class Sleigh implements Runnable {
    public void run() {
        Thread.currentThread().setName("sleighthread");

        while (true) {
            System.out.println("sleigh accepting reindeer");

            // last reindeer awakens sleigh
            reindeerSleighSync.accept();
            System.out.println("sleigh awakened");

            // go to santa and deliver toys
            synchronized (santa) {
                santa.setTask(DELIVERTOYS);
                doorbell.entry();
            }
        }
    }
}

static class Reindeer implements Runnable {
    private int number;

    public Reindeer(int i) {
        number = i;
        }

        public void run() {
            Thread.currentThread().setName("reindeer_thread_" + number);

            while (true) {
                holiday(); // in bahamas
                reindeerSleighSync.entry();
                deliverToys();
            }
        }

        private void holiday() {
            System.out.println("reindeer " + number + " on vacation");
            sleep(1, 3);
            System.out.println("reindeer " + number + " back from vacation");
        }

        private void deliverToys() {
            // wait for santa to unharness
            reindeerSantaSync.entry();
        }
    }

    static class ElfQueue implements Runnable {
        public void run() {
            Thread.currentThread().setName("elfqueue_thread");

            while (true) {
                System.out.println("elfqueue accepting elves");

                // wait for three elves
                elvesQueueSync.accept();
                System.out.println("three elves have a problem");

                synchronized (santa) {
                    santa.setTask(CONSULTELVES);
                    doorbell.entry();
                }
            }
        }
    }

    static class Elf implements Runnable {
        private int number;

        public Elf(int i) {
            number = i;
        }

        public void run() {
            Thread.currentThread().setName("elf_thread_" + number);

            while (true) {
                work(); // until a problem is found
                elvesQueueSync.entry(); // join a group of elves
                consultWithSanta(); // until problem solved
            }
        }

        private void work() {
            System.out.println("elf " + number + " is working");
            sleep(1, 20); // work for a long time
            System.out.println("elf " + number + " found a problem");
        }

        private void consultWithSanta() {
            elvesSantaSync.entry(); // wait for santa to let us out
            System.out.println("elf " + number + " has solved the problem");
        }
    }

    private static int smallRandomInteger(int min, int max) {
        return ((int) (Math.random() * (max + 1 − min))) + min;
    }


    @SuppressWarnings("static-access")
    private static void sleep(int min, int max) {
        try {
            Thread.currentThread().sleep(1000 * smallRandomInteger(min, max));
        } catch (InterruptedException e) {
            // ...
        }
    }
}
```

## A.4 Semaphores

### A.4.1 BinarySemaphore.java

```
public class BinarySemaphore {
    private Coroutine up, down;
    private boolean value;

    public void up() { Coroutine.call(up); }
    public void down() { Coroutine.call(down); }

    public BinarySemaphore() {
        this(true);
    }

    public BinarySemaphore(boolean initial) {
        value = initial;
        up = new BinSem(true);
        down = new BinSem(false);
        up.other = down;
        down.other = up;
    }

    private class BinSem extends AttachedOnly {
        private boolean pval;
        public Coroutine other;
        public BinSem(boolean pval) {
            this.pval = pval;
        }

        public void attachedBody() {
            while (true) {
                if (pval) {
                    passivate();
                }

                value = pval;
                kick(other);
                detach();
            }
        }
    }
}
```

### A.4.2 GeneralSemaphore.java

```
public class GeneralSemaphore {
    private Coroutine up, down;
    private int value;

    public void up() { Coroutine.call(up); }
    public void down() { Coroutine.call(down); }

    public GeneralSemaphore(int initial) {
        value = initial;
        down = new Down();
        up = new Up(down);
    }

    private class Up extends SpinningCoroutine {
        public Coroutine down;
        public Up(Coroutine down) {
            this.down = down;
        }

        public void spinningBody() {
            ++value;
            kick(down);
        }
    }

    private class Down extends AttachedOnly {
        public void attachedBody() {
            while (true) {
                if (value < 1) {
                    passivate();
                }

                --value;
                detach();
            }
        }
    }
}
```

# Bibliography

[1] Ken Arnold and James Gosling. *The Java Programming Language, Second Edition*. Addison-Wesley, second edition, 1998.

[2] Dag Belsnes and Bjarte M. Østvold. Mixing threads and coroutines, *draft*. 2005.

[3] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. In B. Magnusson, editor, *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, 2002.

[4] Nick Benton. Jingle Bells: Solving the Santa Claus Problem in Polyphonic C#, *unpublished*. 2003.

[5] Knut Erik Borgen. Kjøring av Simula i Java-omgivelser. Fra Cim til Jim. Master's thesis, University of Oslo, Departement of Informatics, 2000.

[6] Denis Caromel, Luis Mateu, and Éric Tanter. Sequential object monitors. In Martin Odersky, editor, *ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 316–340. Springer, 2004.

[7] Arthur Charlesworth. The multiway rendezvous. *ACM Trans. Program. Lang. Syst.*, 9(3):350–366, 1987.

[8] Melvin E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, 1963.

[9] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. Simula 67: Common base language. Technical Report 743, Norwegian Computing Center, 1984.

[10] Ana Lúcia de Moura and Roberto Ierusalimschy. Revisiting coroutines. Technical Report MCC15/04, Computer Science Department, Catholic University of Rio de Janeiro, Brazil, 2004. Available at http://www.inf.puc-rio.br/~roberto/docs/MCC15-04.pdf.

[11] Ana Lúcia de Moura, Noemi Rodriguez, and Roberto Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, 2004.

[12] Ana Lucia de Moura, Noemi Rodriguez, and Roberto Ierusalimschy. Coroutines in Lua. 2004.

[13] Edsger W. Dijkstra. Co-operating sequential processes. In *Programming Languages*. Academic Press, 1965.

[14] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.

[15] Allen B. Downey. *The Little Book of Semaphores*. Green Tea Press, second edition, 2005.

[16] C# language specification (ECMA-334). Technical report, ECMA International, 2005.

[17] Common language infrastructure (CLI) (ECMA-335). Technical report, ECMA International, 2005.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995. GAM e 95:1 1.Ex.

[19] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005.

[20] Ralph E. Griswold, David R. Hanson, and John T. Korb. Generators in Icon. *ACM Trans. Program. Lang. Syst.*, 3(2):144–161, 1981.

[21] Dick Grune. A view of coroutines. *SIGPLAN Not.*, 12(7):75–81, 1977.

[22] Tor Hauge, Inger Nordgard, Dan Oscarsson, and Georg Raeder. Event-driven user interfaces based on quasi-parallelism. In *UIST '88: Proceedings of the 1st annual ACM SIGGRAPH symposium on User Interface Software*, pages 66–76, New York, NY, USA, 1988. ACM Press.

[23] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 293–298, New York, NY, USA, 1984. ACM Press.

[24] Keld Helsgaun. Discrete event simulation in Java. Datalogiske skrifter, Roskilde University, 2000.

[25] Richard Kelsey and William Clinger et al. Revised5 report on the algorithmic language Scheme. Technical report, http://www.schemers.org/, 1998.

[26] Donald E. Knuth and Frank Ruskey. Efficient coroutine generation of constrained gray sequences. In *Essays in Memory of Ole-Johan Dahl*, pages 183–208, 2004.

[27] Donald E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, 1974.

[28] Donald E. Knuth. *The Art of Computer Programming, Fundamental Algorithms*, volume 1. Addison-Wesley, third edition, 1997.

[29] Stein Krogdahl. The birth of Simula. In *HiNC 1 Conference*, 2003.

[30] Doug Lea. *Concurrent Programming Java. Design Principles and Patterns*. Addison-Wesley, second edition, 2000.

[31] Doug Lea. Java specification request 166: Concurrency utilities. Technical report, Java Community Process, 2004.

[32] Tim Lindholm and Frank Yelin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[33] Barbara Liskov, Alan Snyder, Russell R. Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Commun. ACM*, 20(8):564–576, 1977.

[34] Christopher D. Marlin. *Coroutines. A Programming Language Methodology, a Language Design and an Implementation*. Number 95 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[35] Robin Milner. *Communicating and Mobile Systems: The $\pi$-calculus*. Cambridge University Press, 1999.

[36] Kristen Nygaard and Ole-Johan Dahl. The development of the simula languages. In *HOPL-1: The first ACM SIGPLAN conference on History of programming languages*, pages 245–272, New York, NY, USA, 1978. ACM Press.

[37] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 138–152, April 2003.

[38] Neil Schemenauer, Tim Peters, and Magnus Lie Hetland. Simple generators. Python Enhancement Proposal (PEP) 255, http://www.python.org/peps/pep-0255.html, 2001.

[39] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, second edition, 2001.

[40] Eric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: spatial and temporal selection of reification. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 27–46, New York, NY, USA, 2003. ACM Press.

[41] John A. Trono. A new exercise in concurrency. *SIGCSE Bull.*, 26(3):8–10, 1994.

[42] Guido van Rossum and Phillip J. Eby. Coroutines via enhanced generators. Python Enhancement Proposal (PEP) 342, http://www.python.org/peps/pep-0342.html, 2005.

[43] Shaula Yemini and Daniel M. Berry. A modular verifiable exception handling mechanism. *ACM Trans. Program. Lang. Syst.*, 7(2):214–243, 1985.