# A Verified Integration of Imperative Parallel Programming Paradigms in an Object-Oriented Language

Thesis by
Paul Sivilotti

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

California Institute of Technology
Pasadena, California

1993
(submitted 30 June 1993)

Caltech CS-TR-93-21

## Abstract

CC++ is a parallel object-oriented programming language that uses parallel composition, atomic functions, and single-assignment variables to express concurrency. We show that this programming paradigm is equivalent to several traditional imperative communication and synchronization models, namely: semaphores, monitors, and asynchronous channels. A collection of libraries which integrates these traditional models with CC++ is specified, implemented, and formally verified.

# Acknowledgments

Many thanks to Mani Chandy whose guidance and encouragement as my research advisor was very much appreciated.

Also thank you to the other members of the Compositional Systems Research Group – Ulla Binau, Pete Carlin, Carl Kesselman, Tal Lancaster, Berna Massingill, Marc Pomerantz, Adam Rifkin, Mei Su, and John Thornley – and to Diana Finley; through valuable discussions, insightful suggestions, or proofreading, they all contributed to this research.

This research was supported in part by NSERC and by ARPA grant N00014-91-J-4014.

# Contents


**1 Introduction** **1**

1.1 Goal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.3 Library Support . . . . . . . . . . . . . . . . . . . . . . . . 3

1.4 Single-Assignment Variables . . . . . . . . . . . . . . . . . . . 4

1.5 CC++ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.6 Organization . . . . . . . . . . . . . . . . . . . . . . . . . . 6

**2 Semaphores** **8**

2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.2 Specification . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.3 Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.3.1 Semaphore as a Class . . . . . . . . . . . . . . . . . . . 9

2.3.2 Suspension and Awakening . . . . . . . . . . . . . . . . 9

2.4 Implementation State and Properties . . . . . . . . . . . . . . 10

2.4.1 State . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.4.2 Properties . . . . . . . . . . . . . . . . . . . . . . . . 11

2.5 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.6 Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

2.6.1 Barrier Synchronization . . . . . . . . . . . . . . . . . 13

2.6.2 Dining Philosophers . . . . . . . . . . . . . . . . . . . 14

2.6.3 Readers Writers . . . . . . . . . . . . . . . . . . . . . 15

2.6.4 Bounded Buffer . . . . . . . . . . . . . . . . . . . . . . 16

2.7 Correctness . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.8 Proof of Specification . . . . . . . . . . . . . . . . . . . . . . . 21

**3 Monitors** **23**
3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.2 Specification . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.3 Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.3.1 Mutual Exclusion . . . . . . . . . . . . . . . . . . . . . 25
3.3.2 Signaling and Waiting . . . . . . . . . . . . . . . . . . 26
3.3.3 Monitor as a Base Class . . . . . . . . . . . . . . . . . 27
3.3.4 Data Encapsulation . . . . . . . . . . . . . . . . . . . 27
3.3.5 Using the Monitor Library . . . . . . . . . . . . . . . 27
3.4 Implementation State and Properties . . . . . . . . . . . . . . 28
3.4.1 State . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.4.2 Properties . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.5 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.6 Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.6.1 Critical Section . . . . . . . . . . . . . . . . . . . . . . 31
3.6.2 Dining Philosophers . . . . . . . . . . . . . . . . . . . 31
3.6.3 Readers Writers . . . . . . . . . . . . . . . . . . . . . 32
3.6.4 Bounded Buffer . . . . . . . . . . . . . . . . . . . . . . 34
3.7 Correctness . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
3.8 Proof of Specification . . . . . . . . . . . . . . . . . . . . . . . 42
3.9 Interpretation of Grammar . . . . . . . . . . . . . . . . . . . . 44
3.10 Confidence in Specification . . . . . . . . . . . . . . . . . . . 45

**4 Asynchronous Channels** **54**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.2 Specification . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.3 The Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
4.3.1 Asynchronous Channel as a Class . . . . . . . . . . . . 55
4.3.2 Messages and Slots . . . . . . . . . . . . . . . . . . . . 55
4.4 Implementation State and Properties . . . . . . . . . . . . . . 57

4.4.1 State . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.4.2 Properties . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.5 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.6 Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
4.6.1 Producer/Consumer . . . . . . . . . . . . . . . . . . . . 59
4.6.2 Dirichlet's Problem . . . . . . . . . . . . . . . . . . . . 60
4.6.3 Mutual Exclusion with a Token Ring . . . . . . . . . . 63
4.7 Correctness . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
4.8 Proof of Specification . . . . . . . . . . . . . . . . . . . . . . 71

**5 A Transport Layer for Ported Asynchronous Channels 72**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
5.2 Specification . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
5.3 Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
5.3.1 Channel as a Pair of Ports . . . . . . . . . . . . . . . . 73
5.3.2 Message Transfer by RPC . . . . . . . . . . . . . . . . 74
5.3.3 Local Copy of Messages . . . . . . . . . . . . . . . . . 74
5.3.4 Performance Considerations . . . . . . . . . . . . . . . 75
5.3.5 Access to `Untransported` Queue . . . . . . . . . . . . 76
5.4 Implementation State and Properties . . . . . . . . . . . . . . 77
5.4.1 State . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
5.4.2 Properties . . . . . . . . . . . . . . . . . . . . . . . . . 78
5.5 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . 79
5.6 Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
5.6.1 Producer/Consumer . . . . . . . . . . . . . . . . . . . . 80
5.7 Correctness . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.8 Proof of Specification . . . . . . . . . . . . . . . . . . . . . . 84

**6 Conclusion 87**

# List of Figures


2.1 Queue of Suspended `P()` Operations . . . . . . . . . . . . . . . 10

3.1 State Transitions for Monitor Processes . . . . . . . . . . . . . 26
3.2 Valid Monitor States . . . . . . . . . . . . . . . . . . . . . . . . . 51
3.3 Valid Monitor State Transitions . . . . . . . . . . . . . . . . . . 53

4.1 Channel Slots and Messages . . . . . . . . . . . . . . . . . . . . 56
4.2 Queue of Empty Slots . . . . . . . . . . . . . . . . . . . . . . . 56
4.3 Queue of Undelivered Messages . . . . . . . . . . . . . . . . . . 57
4.4 Grid for Dirichlet's Problem . . . . . . . . . . . . . . . . . . . . 62
4.5 Token Ring Structure for Mutual Exclusion . . . . . . . . . . . 63
4.6 Multiple Writers to a Single Channel . . . . . . . . . . . . . . . 66

5.1 Division of Channel into Ports . . . . . . . . . . . . . . . . . . . 74
5.2 Local Copy of Message . . . . . . . . . . . . . . . . . . . . . . . 75
5.3 Time Cost Associated with Transport Layer . . . . . . . . . . . 76
5.4 Interaction Between Sending and Transporting Processes . . . 77

# Chapter 1

# Introduction

## 1.1 Goal

My goal is to demonstrate that the single-assignment paradigm is consistent with traditional imperative synchronization and communication schemes, and furthermore that an object-oriented language which incorporates single-assignment variables, atomic functions, and parallel composition can support and seamlessly integrate these methodologies with object orientation. This support and integration is demonstrated by providing a collection of generic libraries whose correctness is formally verified.

## 1.2 Motivation

Parallelism has recently and rapidly become a "truly attractive and viable approach to the attainment of very high computational speeds" [1]. This revolution has been driven by the observation that parallelism is often a natural part of a computation, and that by exploiting this parallelism dramatic speedups can be achieved. With the proliferation of massively parallel architectures for supercomputing, of multiprocessor workstations for multitasking, and of networks of PC's for communication and multiuser applications, parallel programming now stands poised to enter the mainstream of computation.

The problem then becomes: how do we write software for these new architectures? It seems that programming these new parallel architectures is much more difficult than sequential programming in the von Neumann model. Most importantly, verification of parallel imperative programs is believed to be a much harder problem than verification of their sequential counterparts. Many sets of synchronization and communication primitives and programming methodologies have been developed to express parallelism, originally in the context of operating systems, and now in the context of parallel user programs.

Which of these constructs is the "best"? Most of these constructs can be shown to be equivalent in their expressive power, and this equivalence result is commonly seen to support the view that it is sufficient to provide any one paradigm. This work is motivated by the observation that the integration of more than one paradigm in a programming language is desirable, despite this equivalence result. That is, a programmer might wish to use a particular set of primitives based on concerns other than expressiveness.

The integration of a collection of imperative synchronization and communication paradigms in a single programming language is important for four reasons:

- One particular methodology is often most appropriate (perhaps efficient or natural) for a given problem instance. For example: semaphores might be used to program an n-process barrier; a monitor might be used to control access to a bounded buffer; single-assignment variables might be used for a functional implementation of a merge sort; asynchronous channels with explicit message passing might be used to implement a distributed dining philosophers arbitration layer.

- A single problem instance might lend itself to a decomposition in which different components are expressed in different paradigms. For example, an arbitration layer written with explicit message passing might be used to link a physically distributed collection of shared memory MIMD tasks, each of which uses semaphores for internal synchronization and single-assignment variables for functional composition.

- Programmers often become comfortable with a particular methodology. Their experience with that methodology is an important asset, and they should have the flexibility to make use of it.

- There is a wealth of existing programs implemented in a variety of paradigms. It should be possible to reap the benefits of the portability of a language without paying the penalty of excessive modification to existing code.

A parallel programming language should therefore support and integrate a variety of styles and methodologies.

Implementing the synchronization and communication constructs required in a particular instance is often difficult. The correctness of such an implementation is usually not readily apparent from the code, and lengthy and complicated formal arguments are required to convince ourselves that the code does what it should. Clearly such constructs should be implemented and verified once, then the code and proof reused whenever appropriate.

CC++ enables a programmer to make use of a variety of traditional synchronization and communication primitives, in that it is possible to implement these primitives, but this implementation and its verification requires a considerable amount of skill and effort. This work is motivated by the belief that these primitives should be supported in CC++. That is, it should be easy and convenient for a programmer to make use of these methodologies and their formal verifications.

## 1.3 Library Support

Class libraries are used to achieve the desired integration of parallel programming paradigms. C++ provides several powerful mechanisms for library support, including generic classes, inheritance, and separate module compilation and archival. Support for these paradigms is provided at the library level because:

- It does not require making extensions to the language itself. Such extensions require new compilers.
- Abstraction techniques facilitate the integration of well-designed class libraries.

The worth of a library can be measured in several ways. The aims of our design have been:

**Robustness:** is formally verified or, at the very least, thoroughly tested.

**Flexibility:** is usable in a variety of roles and circumstances.

**Generality:** is applicable to a large class of problem instances.

**Extensibility:** is useful as a bottom layer upon which higher level abstractions can be developed.

**Naturalness:** works the way one would expect.

**Expressiveness:** has all the necessary functionality.

**Efficiency:** contains optimizations which make it inexpensive to use.

## 1.4 Single-Assignment Variables

Single-assignment variables are a mechanism for synchronization and communication among concurrent processes. Single-assignment variables can be seen as delayed-assignment constants. The variable is initially undefined, and it can be written to (or *defined*) at most once. A subsequent attempt to write to the variable is a run-time error. If a process attempts to read a single-assignment variable which has not yet been defined, that process suspends until the variable becomes defined [6, Section 2.1].

Single-assignment variables are no more powerful than semaphores. This can be seen by the following C++ implementation of a single-assignment variable generic object, assuming the availability of a semaphore object.

```
template <class T>
class Single <T>  {
  private:
    Semaphore Readers(0);    //Semaphore with initial value 0
    Semaphore Mutex(1);      //Semaphore with initial value 1
    Boolean   Defined;       //has variable been defined?
    T         value;         //value of single-assignment variable
  public:
    Single(void)  {          //Constructor
      Defined = FALSE;
    }

    T read(void)  {
      Readers.P();
      Readers.V();
      return value;
    }

    void write(T v)  {
      Mutex.P();
      if (Defined == TRUE)  {
        Mutex.V();
```

```
        write_twice_error_condition;
      else  {
        Defined = TRUE;
        value = v;
        Mutex.V();
        Readers.V();
      }
    }
};
```

Thus, single-assignment variables can be seen as simply a discipline on using semaphores.

## 1.5 CC++

CC++ is a parallel object-oriented programming language based on C++. It makes use of single-assignment variables, atomic functions, and parallel composition to express and control concurrency. The following two CC++ constructs are used in the implementation of the semaphore, monitor, and asynchronous channel libraries presented in chapters 2-4:

`sync`: a single-assignment object.

`atomic`: a function whose statements are not interleaved with statements that are not part of the function.

In addition, the transport layer optimization presented in chapter 5, and the example programs used through out the report, also make use of the following CC++ constructs:

`par`: a block of statements which are executed in parallel.

`parfor`: a loop whose iterations are executed in parallel.

`spawn`: the creation of a new thread of control which executes in parallel with the spawning thread.

`global`: a pointer which spans address spaces.

For a complete description of these constructs, refer to [12] and [15].

A queue library is also used. This is a C++ queue with the following public interface:

```
template <class T>
class Queue<T>  {
  public:
    void enqueue(T*);      //adds one item to queue
    T * dequeue(void);     //removes first item from queue
    int isempty(void);     //returns 1 if queue empty (0 otherwise)
};
```

## 1.6 Organization

In addition to the introduction and conclusion, this report is divided into 4 chapters, each of which describes a library to support a different set of synchronization and communication primitives. Each chapter is divided into the following sections:

**Introduction:** an intuitive description of the primitive being supported.

**Specification:** a formal specification of the primitive.

**Design:** a description of the considerations and decisions that were made when designing the implementation.

**Implementation State and Properties:** the invariants that characterize the implementation. The conjunction of these invariants imply the specification.

**Implementation:** the code of the library.

**Examples:** several examples of how the library can be used to solve some classic synchronization problems.

**Correctness:** the implementation annotated with assertions to prove that the invariants given in "Implementation State and Properties" are maintained.

**Proof of Specification:** proof that the invariants given in the "Implementation State and Properties" section imply the specification.

The following points concerning the assertional annotation of these libraries should be noted:

- An invariant need not hold in the middle of an atomic action. The invariants must only be shown to hold at the beginning and end of such an action.
- Appending to a queue increments the queue size by 1, while dequeuing decrements the queue size by 1.
- Auxiliary variables are used to simplify the assertions. Changes to these auxiliary variables are shown in *emphasis* font.

# Chapter 2

# Semaphores

## 2.1 Introduction

A semaphore is a mechanism which provides a means for synchronization between concurrent processes. A semaphore has an integer value which is invariantly non-negative. Three operations are defined for a semaphore: initialization, `P()`, and `V()`.

- Initialization sets the value of the semaphore to a non-negative integer.
- The `V()` operation atomically increments this value. It never suspends.
- The `P()` operation attempts to decrement the semaphore's value. If the value of the semaphore is greater than 0, then the decrement is performed and the `P()` operation terminates. If, on the other hand, the value of the semaphore is 0 (and therefore a decrement would violate the invariant that the value be non-negative), then the `P()` operation suspends until the decrement may be safely performed.

Semaphores were introduced in [7] and are further discussed in [9], [14], and [2, Chapter 4].

## 2.2 Specification

Since the `P()` operation is not atomic, we distinguish between the initiation and the termination of this operation. Let `iP` be the number of `P()` operations that have been *initiated*, and let `cP` be the number of `P()` operations that have *completed*. Since the `V()` operation is atomic, no such distinction is required and we define `cV` to be the number of completed `V()` operations.

The initial value of the semaphore is defined to be $\mathtt{s}_0$ where $\mathtt{s}_0 \geq 0$. The safety condition is that as many `P()` operations as possible will complete, subject to the constraints of the number of initiated `P()` operations and the sum of the initial value of the semaphore and the number of completed `V()` operations.

$$\mathtt{cP} = \min(\mathtt{iP}, \mathtt{s}_0 + \mathtt{cV}) \tag{2.1}$$

In addition, our implementation of semaphores meets the fairness requirement that the $i^{th}$ `P()` operation initiated can be allowed to terminate only if the $(i-1)^{th}$ `P()` operation has also been allowed to terminate. Denoting the $(i+1)^{th}$ `P()` operation by $\mathtt{P}_i$ (for $i \geq 0$) we have:

$$\forall i : i \geq 1 : \mathtt{terminated}(\mathtt{P}_i) \Rightarrow \mathtt{terminated}(\mathtt{P}_{i-1}) \tag{2.2}$$

## 2.3 Design

### 2.3.1 Semaphore as a Class

A semaphore can be implemented as a CC++ class. It stores its current value in a private data member `value`. Access is via two public member functions, `P()` and `V()`. Since the `V()` member function never suspends, and since it manipulates the shared mutable `value`, this function is atomic. The member function `P()`, on the other hand, can suspend, and so is not atomic. The class also contains a queue to keep track of the suspended `P()` operations.

### 2.3.2 Suspension and Awakening

To control the suspension and awakening of `P()` operations, pointers to `sync` objects are used. Since these objects are used only for this purpose, a unary type suffices. This `sync` object then either can be undefined, or it can be defined to the unary value permitted by the type. For our purposes we use the unary type which permits only the value `SET`.

A `P()` operation begins by creating a pointer, called `stop`, to a new `sync` object of our unary type. If the value of the semaphore is 0, then

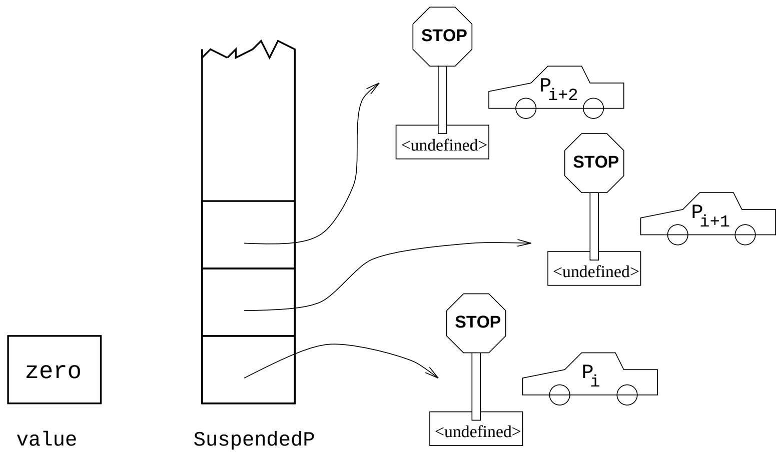


Figure 2.1: Queue of Pointers to `sync` Objects for Suspending `P()` Operations

this pointer is added to the `SuspendedP` queue. The `P()` operation then suspends on the contents of `stop` (see Figure 2.1).

A `V()` operation begins by checking whether there are any suspended `P()` operations. If there are, then the first element of `SuspendedP` is dequeued and the `sync` value defined (to `SET`). If there are not, then the value of the semaphore is incremented.

## 2.4 Implementation State and Properties

### 2.4.1 State

The state of a semaphore is given by:

1. The value of the semaphore, `s`. Since each completed `P()` operation decrements this value and each completed `V()` operation increments this value, we have:

$$\mathtt{s} = \mathtt{s}_0 - \mathtt{cP} + \mathtt{cV} \tag{2.3}$$

2. The queue of suspended `P()` operations. The number of suspended `P()` operations, `qP`, is the number of `P()` operations which have been initiated, but have not yet completed.

$$\mathtt{iP} = \mathtt{qP} + \mathtt{cP} \tag{2.4}$$

Also, the queue maintains a FIFO ordering of these elements.

$$\forall j : 0 \leq j < \mathtt{qP} : \mathtt{SuspendedP}[j] = \mathtt{P}_{\mathtt{cP}+j} \tag{2.5}$$

Here the queue is SuspendedP and SuspendedP[$n$] denotes the $(n+1)^{th}$ element of the queue.

A P() operation that has been initiated can be in one of two possible states: **suspended** or **terminated**. A P() operation is defined to be suspended precisely when it is an element of the SuspendedP queue. It is terminated if and only if it has been initiated and is not suspended.

$$\forall i : 0 \leq i < \mathtt{iP} : \mathtt{terminated}(\mathtt{P}_i) \Leftrightarrow \mathtt{initiated}(\mathtt{P}_i) \wedge \neg \mathtt{suspended}(\mathtt{P}_i) \tag{2.6}$$

### 2.4.2 Properties

**Boundedness Requirement.** The number of P() operations that can complete is bounded by the initial value of the semaphore and the number of V() operations that have completed.

$$\mathtt{cP} \leq \mathtt{s}_0 + \mathtt{cV} \tag{2.7}$$

**The Set of Suspended Processes is Minimal.** A process can be suspended only if its completion would violate the safety condition given by (2.1).

$$(\mathtt{qP} = 0) \vee (\mathtt{s} = 0) \tag{2.8}$$

## 2.5 Implementation

```
typedef enum {SET=1} Unary;

class Semaphore {
  private:
    int value;                                    //value of the semaphore
    Queue<sync Unary> SuspendedP;                 //queue for susp. P ops

    atomic void check_value (sync Unary * ptr);
```

```
  public:
    Semaphore (int);
    atomic void V(void);
    void P(void);
};
```

**Constructor**

```
Semaphore::Semaphore (int v=1) {
  if (v ≥ 0) value = v;
  else error_condition;
}
```

**Check_Value**

```
atomic void Semaphore::check_value (sync Unary * ptr) {
  if (value > 0) {
    value--;
    *ptr = SET;
  }
  else {
    SuspendedP.enqueue(ptr);
  }
}
```

**V**

```
atomic void Semaphore::V(void) {
  if (SuspendedP.isempty()) value++;
  else *(SuspendedP.dequeue()) = SET;
}
```

**P**

```
void Semaphore::P(void) {
  sync Unary * stop = new sync Unary;
  check_value(stop);
  if (*stop == SET) delete stop;
}
```

## 2.6 Examples

### 2.6.1 Barrier Synchronization

**Motivation** Our first example is a simple demonstration of how semaphores are declared, initialized, and used.

**Problem Description** Two concurrent processes wish to repeatedly synchronize at a certain point in their computations. Neither process should be allowed past this point before the other process has reached the same point. This is known as a barrier.

```
#define N 20                                    //number of iterations

Semaphore barrier1(0);
Semaphore barrier2(0);

void P1(void) {
  for (int i=0; i<N; i++) {
    work();
    barrier1.V();
    barrier2.P();
  }
}

void P2(void) {
  for (int i=0; i<N; i++) {
    work();
    barrier2.V();
    barrier1.P();
  }
}

main()
{
  par {
    P1();
    P2();
  }
}
```

**Discussion** The semaphores `barrier1` and `barrier2` are initialized with the value 0. The semaphore operations `P()` and `V()` use the regular C++ member function notation, for example `barrier1.V()`.

### 2.6.2 Dining Philosophers

**Motivation** Arrays of semaphores can be declared and initialized. Semaphores should be passed to functions as reference parameters.

**Problem Description** Five philosophers are sitting around a table. There is a fork between each one. Philosophers cycle between *thinking* and *eating*. In order to eat, a philosopher needs both the fork to his left and the fork to his right. A fork cannot be simultaneously held by two people.

```
#define N 20                              //number of iterations

void Philosopher (Semaphore &left, Semaphore &right) {
  for (int i=0; i<N; i++) {
    left.P(); right.P();
    eat();
    left.V(); right.V();
    think();
  }
}

main()
{
  Semaphore fork[5] = {1,1,1,1,1};
  par {
    parfor (int i=0; i≤3; i++) Philosopher(fork[i],fork[i+1]);
    Philosopher(fork[0],fork[4]);
  }
}
```

**Discussion** Arrays of semaphores can be declared and each semaphore initialized with its own value. In this case, `fork` is an array of 5 semaphores, each of which is initialized with the value 1. One must be careful when passing semaphores to functions. Because of the pass-by-value semantics of regular argument evaluation, semaphores must be passed by reference. Otherwise, the function will see only a local copy of the semaphore and its `P()` and `V()` operations will have no visible effect outside the function. Note that `left` and `right` are declared in the header of `Philosopher()` to be references to semaphores.

### 2.6.3 Readers Writers

**Motivation** Semaphores can be members. The default constructor initializes a semaphore's value to 1.

**Problem Description** A collection of concurrent processes share a database. Some processes, the *readers*, wish to read data, while other processes, the *writers*, wish to modify the data. Many readers can access the database concurrently, but each writer needs exclusive access.

```
class Database {
  private:
    int nr;                                        //number of readers
    Semaphore mutexR;                //mutual exclusion between readers
    Semaphore rw;               //mutual exclusion for accessing database

  public:
    Database (void)         //mutexR and rw implicitly initialized to 1
      { nr = 0; }

    void reader (void) {
      mutexR.P();
        nr += 1;
        if (nr == 1) rw.P();
      mutexR.V();
      read();                                      //read the database
      mutexR.P();
        nr -= 1;
        if (nr == 0) rw.V();
      mutexR.V();
    }

    void writer (void) {
      rw.P();
      write();                                     //write the database
      rw.V();
    }
};
```

**Discussion** The default constructor for semaphores initializes the value to 1. If this is the desired initial value (as is the case in the `Database` class), no explicit construction of the member is required. This practice, however, is not encouraged since it makes the initial value implicit.

### 2.6.4 Bounded Buffer

**Motivation** Member semaphores can be initialized with explicit construction.

**Problem Description** A bounded buffer is a multislot communication buffer. Processes known as *producers* add messages to the buffer, while processes known as *consumers* remove them. A producer may insert a message if there is at least one empty slot. A consumer may remove a message if there is at least one full slot. Insertions must be mutually exclusive to preserve the integrity of the buffer, as must deletions be.

```
class BoundedBuffer {
  private:
    int size;
    char *Buf;
    int front, rear;
    Semaphore empty, full;   //assert: size-2 <= empty+full <= size
    Semaphore mutexD, mutexF;

  public:
    BoundedBuffer(int n) :  empty(n), full(0),
                           mutexD(1), mutexF(1) {
      size = n;
      Buf = new char[size];
      front = 0;
      rear = 0;
    }

    void deposit (char data) {
      empty.P();
      mutexD.P();
        buf[rear] = m; rear = (rear+1)%size;
      mutexD.V();
      full.V();
    }

    void fetch (char &data) {
      full.P();
      mutexF.P();
        data = buf[front]; front = (front+1)%size;
      mutexF.V();
      empty.V();
    }
};
```

**Discussion** Here the semaphore members `empty` and `full` must be initialized to values which are not 1. Hence, the semaphores must be explicitly constructed. This is accomplished by following the constructor for the enclosing class `BoundedBuffer` with a colon, then the semaphores and their initial values. The semaphores `mutexD` and `mutexF` could have been omitted from this list since their initial values are the same as that provided by the default constructor (*i.e.* 1). This is the usual mechanism provided by C++ for initializing member classes.

## 2.7 Correctness

We show the invariance of equations 2.3 through 2.8 by annotating the text of the implementation. Note that these equations need not hold in the middle of atomic actions, but only at the beginning and at the end of such actions.

### Annotated Program for Constructor

```
Semaphore::Semaphore (int v = 1)
{
  if (v >= 0) value = v;
  // s = s0 = v;
  // cP = cV = qP = iP = 0;

  else error_condition;

  // BEGIN ASSERTIONS
  // s = s0 = v
  // cP = cV = qP = iP = 0
  // (2.3) - (2.8)
  // END ASSERTIONS
}
```

### Annotated Program for Check_Value

```
atomic void Semaphore::check_value(sync Unary * ptr)
{
  // BEGIN ASSERTIONS
```

```
// (2.3) - (2.8)
// END ASSERTIONS

// iP++;

// BEGIN ASSERTIONS
// iP = qP + cP + 1
// initiated(P_{iP-1}) ∧ ¬suspended(P_{iP-1}) ∧ ¬terminated(P_{iP-1})
// (2.3), (2.5), (2.7), (2.8)
// END ASSERTIONS

if (value > 0) {
  // BEGIN ASSERTIONS
  // s > 0
  // qP = 0
  // iP = qP + cP + 1
  // initiated(P_{iP-1}) ∧ ¬suspended(P_{iP-1}) ∧ ¬terminated(P_{iP-1})
  // (2.3), (2.5), (2.7), (2.8)
  // END ASSERTIONS

  value--;
  // s--;

  // BEGIN ASSERTIONS
  // s = s_0 - cP + cV - 1
  // s ≥ 0
  // qP = 0
  // iP = qP + cP + 1
  // initiated(P_{iP-1}) ∧ ¬suspended(P_{iP-1}) ∧ ¬terminated(P_{iP-1})
  // (2.5), (2.7), (2.8)
  // END ASSERTIONS

  *ptr = SET;
  // cP++;

  // BEGIN ASSERTIONS
  // qP = 0
  // terminated(P_{iP-1})
```

```
    // (2.3) - (2.8)
    // END ASSERTIONS

  }
  else {

    // BEGIN ASSERTIONS
    // s = 0
    // iP = qP + cP + 1
    // initiated(P_{iP-1}) ∧ ¬suspended(P_{iP-1}) ∧ ¬terminated(P_{iP-1})
    // (2.3), (2.5), (2.7), (2.8)
    // END ASSERTIONS

    SuspendedP.enqueue(ptr);
    // qP++;

    // BEGIN ASSERTIONS
    // s = 0
    // initiated(P_{iP-1}) ∧ suspended(P_{iP-1})
    // (2.3) - (2.8)
    // END ASSERTIONS
  }
}

// BEGIN ASSERTIONS
// (2.3) - (2.8)
// END ASSERTIONS
```

**Annotated Program for V**

```
atomic void Semaphore::V(void)
{
  // BEGIN ASSERTIONS
  // (2.3) - (2.8)
  // END ASSERTIONS

  if (SuspendedP.isempty())
```

```
    // BEGIN ASSERTIONS
    // qP = 0
    // (2.3) - (2.8)
    // END ASSERTIONS

    value++;
    // s++;

    // BEGIN ASSERTIONS
    // qP = 0
    // s = s_0 - cP + cV + 1
    // (2.4) - (2.8)
    // END ASSERTIONS

  else

    // BEGIN ASSERTIONS
    // qP > 0
    // s = 0
    // SuspendedP[0] = P_cP
    // (2.3) - (2.8)
    // END ASSERTIONS

    *(SuspendedP.dequeue()) = SET;
    // qP--;
    // cP++;

    // BEGIN ASSERTIONS
    // s = 0
    // s = s_0 - cP + cV + 1
    // terminated(P_cP-1)
    // (2.4) - (2.6), (2.8)
    // END ASSERTIONS

}
// cV++;

// BEGIN ASSERTIONS
```

```
// (2.3) - (2.8)
// END ASSERTIONS
```

**Annotated Program for P**

```
void Semaphore::P(void)
{
  // BEGIN ASSERTIONS
  // (2.3) - (2.8)
  // END ASSERTIONS

  sync Unary * stop = new sync Unary;
  check_value(stop);

  // BEGIN ASSERTIONS
  // (2.3) - (2.8)
  // END ASSERTIONS

  if (*stop == SET)

    // BEGIN ASSERTIONS
    // (2.3) - (2.8)
    // END ASSERTIONS

    delete stop;
}
// BEGIN ASSERTIONS
// (2.3) - (2.8)
// END ASSERTIONS
```

## 2.8 Proof of Specification

**Proof of (2.1)** :

```
TRUE
```

$\Leftrightarrow$ { by 2.8, 2.3, and 2.4 }

$(\mathtt{cP} = \mathtt{iP}) \vee (\mathtt{cP} = \mathtt{s}_0 + \mathtt{cV})$

$\Leftrightarrow$ { by 2.7 and definition of cP }

$(\mathtt{cP} \geq \min(\mathtt{iP}, \mathtt{s}_0 + \mathtt{cV})) \wedge (\mathtt{cP} \leq \mathtt{iP}) \wedge (\mathtt{cP} \leq \mathtt{s}_0 + \mathtt{cV})$

$\Leftrightarrow$ { property of min }

$\mathtt{cP} = \min(\mathtt{iP}, \mathtt{s}_0 + \mathtt{cV}) \quad \Box$

**Proof of (2.2)** : For any $i$ greater than or equal to 1 we have

$\mathtt{terminated}(\mathtt{P}_i) \wedge (i \geq 1)$

$\Leftrightarrow$ { by 2.6 }

$\mathtt{initiated}(\mathtt{P}_i) \wedge \neg\mathtt{suspended}(\mathtt{P}_i) \wedge (i \geq 1)$

$\Leftrightarrow$ { by definition of initiated() and suspended() }

$(i < \mathtt{iP}) \wedge (\mathtt{P}_i \notin \mathtt{SuspendedP}) \wedge (i \geq 1)$

$\Leftrightarrow$ { by 2.5 }

$(1 \leq i < \mathtt{iP}) \wedge (\nexists j : 0 \leq j < \mathtt{qP} : \mathtt{cP} + j = i)$

$\Leftrightarrow$ { by 2.4 }

$1 \leq i < \mathtt{cP}$

$\Rightarrow$ { }

$0 \leq i - 1 < \mathtt{cP}$

$\Leftrightarrow$ { by 2.4 }

$(0 \leq i - 1 < \mathtt{iP}) \wedge (\nexists j : 0 \leq j < \mathtt{qP} : \mathtt{cP} + j = i - 1)$

$\Leftrightarrow$ { by 2.5 }

$(0 \leq i - 1 < \mathtt{iP}) \wedge (\mathtt{P}_{i-1} \notin \mathtt{SuspendedP})$

$\Leftrightarrow$ { by definition of initiated() and suspended() }

$\mathtt{initiated}(\mathtt{P}_{i-1}) \wedge \neg\mathtt{suspended}(\mathtt{P}_{i-1})$

$\Leftrightarrow$ { by 2.6 }

$\mathtt{terminated}(\mathtt{P}_{i-1}) \quad \Box$

# Chapter 3

# Monitors

## 3.1 Introduction

A monitor is a class. It is a collection of data (member data) and of functions (member functions) which manipulate this data. A monitor is shared between concurrent processes and provides synchronization between these processes in two ways:

1. Implicitly: only one process at a time is permitted to be inside a monitor executing any of the member functions.

2. Explicitly: processes can synchronize by means of special variables called *condition variables*.

A monitor guarantees that only its own member functions have access to its member data, and therefore it can be used to implement a critical section which requires exclusive access to a shared resource. In addition, a monitor provides a special type of variable, called a condition variable, and two functions `signal()` and `wait()`, which can be executed on this type of variable. These functions are the only valid operations on a condition variable. They can be used only inside monitor member functions. The operation `wait()` on a condition variable $c$ causes the executing process to be suspended and placed in a pool of blocked processes associated with $c$. The `signal()` operation on condition variable $c$ causes one of the processes (if there are any) in the pool of suspended processes associated with $c$ to become ready to execute.

It is important to note that when a process issues a `signal()` on a condition variable, there are potentially two processes which are now eligible to execute inside the monitor: the process which executed the `signal()` and the process that was selected for reactivation (if there was one). We choose here to allow the signaling process to continue executing inside the monitor. This signaling discipline is known as "signal and continue".

Monitors were introduced in [9] and [8]. They are described in detail in [2] and [4], including extensions such as other possible signaling disciplines, extended primitives, and priority waiting.

## 3.2 Specification

A monitor guarantees mutual exclusion implicitly by allowing only one of its member functions to be executing at a time. Let `#enter` be the number of times processes have entered the monitor (and begun executing) and let `#leave` be the number of times they have left the monitor (either because the function being executed terminated or performed a `wait()`). The safety condition is then:

$$(\#\texttt{enter} = \#\texttt{leave}) \vee (\#\texttt{enter} = \#\texttt{leave} + 1) \tag{3.1}$$

If every member function of a monitor either terminates or executes a wait, then every function that becomes ready to execute is eventually allowed to enter. Unfortunately, programmers can write functions that do not terminate. Hence, only a much weaker progress property can be stated. Let `#became_ready` be the number of processes which became ready to execute (either by calling a monitor function, or by being reawakened after a `wait()`). Then we have:

$$\#\texttt{enter} = \min(\#\texttt{became\_ready}, \#\texttt{leave} + 1) \tag{3.2}$$

We distinguish between the two cases for a `signal()`: either there is at least one process waiting on the signaled condition variable, or there is none. In the former case, the `signal()` is designated as *heard* since it causes a process to become ready (the signaling process does not relinquish the monitor), and in the latter case it is designated as *ignored* since it has no effect. Initially there are no waiting processes. So there are three valid operations on a condition variable: a wait, an ignored signal, and a heard signal. If the number of waits exceeds the number of signals which have been heard, then the next signal will be heard. Thus, if $\Sigma = \{\mathbf{w}, \mathbf{s}_i, \mathbf{s}_h\}$ is the alphabet representing the operations wait, ignored signal, and heard signal respectively, and $V^c \in \Sigma^*$ is the string of operations which have been performed on condition variable $c$, then:

$$\exists s : s \in \mathcal{L}(G) : \quad V^c \leq s \tag{3.3}$$

where $w \leq z$ for two strings $w$ and $z$ denotes that $w$ is a prefix of $z$ and where $G$ is the context-free grammar defined by the start symbol $S$, the set $\Sigma$ of terminals, and the production rules:

$$\begin{aligned} S &\rightarrow \mathbf{s}_i S \mid BS \mid \epsilon \\ B &\rightarrow B\mathbf{w}B\mathbf{s}_h \mid \epsilon \end{aligned}$$

## 3.3 Design

### 3.3.1 Mutual Exclusion

We must guarantee that there is at most one process executing inside the monitor at any one time. This mutual exclusion is provided by two member functions, `enter()` and `leave()`. The `enter()` function is used at the beginning of every user-defined member function and asks permission to enter the monitor. The `leave()` function is used at the end of every user-defined member function and notifies the monitor that another process can now be allowed to enter.

The monitor maintains a private Boolean variable, `busy`, which is `TRUE` exactly when there is a process executing inside. The monitor also maintains a private queue, `Ready`, of processes which are ready to execute, but prevented from entering by the mutual exclusion requirement. This queue is simply a collection of pointers to undefined `sync` objects on which the processes that are ready are suspended. Allowing a process to enter then amounts to dequeuing this pointer and defining the `sync` object.

The rule, then, for using the monitor library is simple: every member function must begin with a call to `enter()` and must terminate with a call to `leave()`. The behavior of a monitor that contains member functions which do not obey this rule is not defined.

### 3.3.2 Signaling and Waiting

The `signal()` and `wait()` operations are performed on condition variables. They are the only valid operations on these variables. Condition variables are visible only within a monitor, so they are implemented as a nested class. With each condition variable a queue is associated. When a process executes a `wait()`, it must exit the monitor and suspend. Exiting the monitor is accomplished by dequeuing the next ready process, if one exists, or by setting the `busy` flag to `FALSE` if one does not. The suspension on a condition variable is accomplished by creating a new (undefined) `sync` object and appending a pointer to it on the condition variable's associated queue. The process then attempts to read the contents of this `sync` object.

When a process executes a `signal()` on condition variable $c$, it must make a process that is waiting on $c$ (if one exists) ready to execute. This is accomplished by checking the queue associated with $c$. If this queue is empty, nothing needs to be done. If the queue is not empty, the first element (which is a pointer to an undefined `sync` object) is dequeued and added to the `Ready` queue. The signaling process remains inside the monitor.

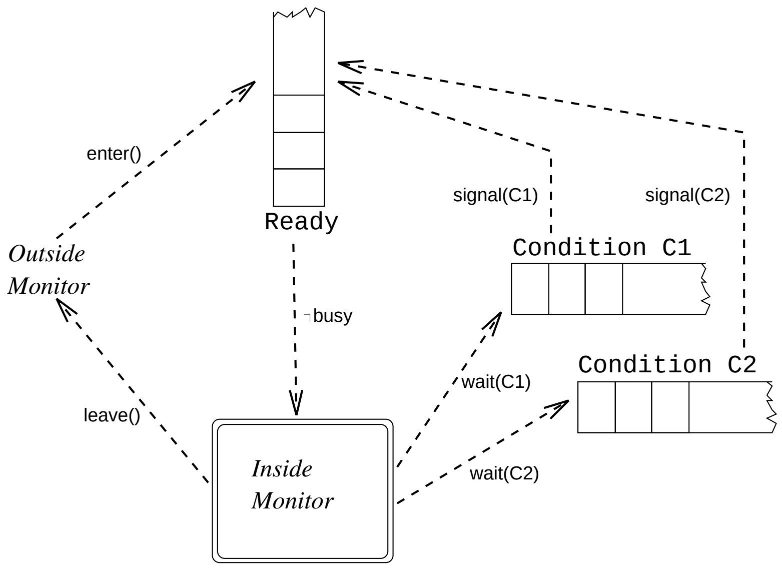


Figure 3.1: State Transitions for Processes using a Monitor

### 3.3.3 Monitor as a Base Class

A monitor can be seen quite naturally as a CC++ class. The concepts of data and function encapsulation of a monitor are consistent with those of CC++ classes. There are certain constructs, specific to monitors, which all monitors share. A monitor contains a definition of a condition variable type, as well as two member functions (`wait()` and `signal()`) to operate on this type. Also, a monitor ensures mutual exclusion between processes with two more member functions `enter()` and `leave()`.

In addition to these constructs which all monitors have, a monitor must contain the particular member functions specific to the situation to which it is being applied. For example, a bounded buffer monitor might have two member functions `insert()` and `remove()` to control access to the buffer. To be useful in the context of a general library, a programmer must be able to "fill in" the member functions required by a particular problem instance. Thus, a particular instantiation of a monitor will be a superset of the fundamental data type and function members which all monitors share.

This functionality can be achieved by implementing the `Monitor` class as a base class. To make use of this library, the programmer will create his or her particular monitor class by deriving from the base `Monitor` class. The programmer may then specify the additional member functions and data that are required.

### 3.3.4 Data Encapsulation

A monitor must guarantee that only its own member functions have access to its member data. C++ has a simple mechanism to ensure this protection: the member data can be declared `private`. Again, the rule is simple: all member data of a monitor must be declared `private`.

Of course, the library implementation of the base `Monitor` class cannot guarantee that all derivations from this base class will have only private member data. The behavior of a monitor with non-private member data is not defined.

### 3.3.5 Using the Monitor Library

An application which uses the `Monitor` library to implement a particular monitor, must obey the following rules:

1. The user-defined monitor must be derived from the library monitor.
2. All member data must be private.
3. Every member function must begin with a call to `enter()`, and must terminate with a call to `leave()`. Note that this implies that the outer block of a monitor member function must be a sequential block. It also implies that no member function may contain a `return` statement, and hence all member functions must be `void` functions.

## 3.4 Implementation State and Properties

### 3.4.1 State

The state of a monitor is defined by:

1. Whether or not a process is inside the monitor. If such a process is executing inside, the Boolean flag `busy` has value `TRUE`, otherwise it has value `FALSE`.

$$\begin{aligned} \texttt{busy} &\Leftrightarrow \#\texttt{enter} = \#\texttt{leave} + 1 && (3.4) \\ \neg\texttt{busy} &\Leftrightarrow \#\texttt{enter} = \#\texttt{leave} && (3.5) \end{aligned}$$

2. The queue (`Ready`) of processes which are ready to execute inside the monitor. The size of this queue is the difference in the number of processes which became ready and the number which entered the monitor.

$$\#\texttt{enter} \leq \#\texttt{became\_ready} \quad (3.6)$$

3. The queues of waiting processes associated with each condition variable. Let $Q^c_k$ be the number of processes waiting on condition variable $c$ after $k$ operations on that variable. Initially, the wait queues are empty, so $Q^c_0 = 0$. Since each `wait(`$c$`)` operation appends a process to this queue and each `signal(`$c$`)` operation that is heard removes one, we have:

$$\begin{aligned} \forall k &: |V^c| \geq k \geq 0 : \quad 0 \leq Q^c_k && (3.7) \\ \forall k &: |V^c| \geq k \geq 1 : \quad Q^c_k = \left|V^c_{0..k-1} \restriction \{\mathbf{w}\}\right| - \left|V^c_{0..k-1} \restriction \{\mathtt{s}_h\}\right| && (3.8) \end{aligned}$$

Since `signal()`'s on empty queues are always ignored and `signal()`'s on non-empty queues are always heard, we have:

$$\forall k \quad : |V^c| > k \geq 0 : \quad Q^c_k = 0 \Rightarrow V^c_k \neq \mathbf{s}_h \tag{3.9}$$

$$\forall k \quad : |V^c| > k \geq 0 : \quad Q^c_k > 0 \Rightarrow V^c_k \neq \mathbf{s}_i \tag{3.10}$$

### 3.4.2 Properties

**Maximality of Progress** As many processes as possible will be allowed to enter the monitor, subject to the constraints of mutual exclusion and of the number which became ready.

$$\texttt{busy} \vee (\#\texttt{enter} = \#\texttt{became\_ready}) \tag{3.11}$$

## 3.5 Implementation

```
typedef enum {SET=1} Unary;
typedef enum {FALSE=0,TRUE=1} Boolean;

class Monitor {
  private:
    Boolean busy;
    Queue<sync Unary> Ready;

    atomic void check_busy(sync Unary *);

  protected:
    Monitor(void);
    void enter(void);
    atomic void leave(void);
    class Condition
      { public:  Queue<sync Unary> Waiting; };
    void wait(Condition&);
    void signal(Condition&);
};
```

**Constructor**

```
Monitor::Monitor(void) {
  busy = FALSE;
}
```

**Check_Busy**

```
atomic void Monitor::check_busy(sync Unary * ptr) {
  if (busy == FALSE) {
    busy = TRUE;
    *ptr = SET;
  }
  else {
    Ready.enqueue(ptr);
  }
}
```

**Enter**

```
void Monitor::enter(void) {
  sync Unary *hold = new sync Unary;
  check_busy(hold);
  if (*hold == SET) delete hold;
}
```

**Leave**

```
atomic void Monitor::leave(void) {
  if (Ready.isempty()) busy = FALSE;
  else *(Ready.dequeue()) = SET;
}
```

**Wait**

```
void Monitor::wait(Condition &c) {
  sync Unary *stop = new sync Unary;
  c.Waiting.enqueue(stop);
  leave();
  if (*stop == SET) delete stop;
}
```

**Signal**

```
void Monitor::signal(Condition &c) {
  if (c.Waiting.isempty() == 0)
    Ready.enqueue(c.Waiting.dequeue());
}
```

## 3.6 Examples

### 3.6.1 Critical Section

**Motivation** Our first example is a simple demonstration of how a monitor can be created by deriving from the base `Monitor` class.

**Problem Description** We have a collection of concurrent processes that require mutually exclusive access to a critical section.

```
class CriticalSection : private Monitor {
  public:
    void access (void) {
      enter();
      critical_section();
      leave();
    }
};
```

**Discussion** The C++ mechanism of inheritance is used to create a monitor. Note how the monitor function `access()` begins with a call to `enter()` and terminates with a call to `leave()`.

### 3.6.2 Dining Philosophers

**Motivation** Arrays of condition variables can be declared inside monitors.

**Problem Description** As described in chapter 2, five philosophers are sitting around a table. There is a fork between each one. Philosophers cycle between *thinking* and *eating*. In order to eat, a philosopher needs both the fork to his left and the fork to his right. A fork cannot be simultaneously held by two people.

```
class DiningRoom :  private Monitor {
  private:
    int forks[5];
    Condition forks_available[5];

  public:
    DiningRoom (void)
      { parfor (int i=0; i≤4; i++) forks[i] = 2; }

    void start_eating(int i) {
      enter();
      if (forks[i] ≠ 2) wait(forks_available[i]);
      forks[(i+4)%5] -= 1;
      forks[(i+1)%5] -= 1;
      leave();
    }

    void stop_eating(int i) {
      enter();
      forks[(i+4)%5] += 1;
      forks[(i+1)%5] += 1;
      if (forks[(i+4)%5] == 2) signal(forks_available[(i+4)%5]);
      if (forks[(i+1)%5] == 2) signal(forks_available[(i+1)%5]);
      leave();
    }
};
```

**Discussion** Note how the data members of a monitor (the arrays `forks` and `forks_available`) are private members. The constructor is used to initialze these members. Arrays of condition variables are declared and used in the usual CC++ manner.

### 3.6.3 Readers Writers

**Motivation** The outer block of member functions must be sequential.

**Problem Description** As described in chapter 2, a collection of concurrent processes share a database. Some processes, the *readers*, wish to read data, while other processes, the *writers*, wish to modify the data. Many readers can access the database concurrently, but each writer needs exclusive access.

```
class RW_Controller :  private Monitor {
```

```
  private:
    int nr, nw;
    Condition oktoread;                    //signaled when nw = 0
    Condition oktowrite;          //signaled when nr = 0 ∧ nw = 0

  public:
    RW_Controller(void) {
      nr = 0; nw = 0;
    }

    void request_read(void) {
      enter();
      while (nw > 0) wait(oktoread);
      signal(oktoread);
      nr = nr+1;
      leave();
    }

    void release_read(void) {
      enter();
      nr = nr-1;
      if (nr == 0) signal(oktowrite);
      leave();
    }

    void request_write(void) {
      enter();
      while ((nr > 0) || (nw > 0)) wait (oktowrite);
      nw = nw+1;
      leave();
    }

    void release_write(void) {
      enter();
      par {
        nw = nw-1;
        signal(oktowrite);
        signal(oktoread);
      }
      leave();
    }
};
```

**Discussion** The body of the `release_read()` member function can be a parallel block. However, `enter()` and `leave()` must be the first and last statements, so the outermost block must be sequential.

### 3.6.4 Bounded Buffer

**Motivation** Monitor member functions must be `void` functions.

**Problem Description** As described in chapter 2, a bounded buffer is a multislot communication buffer. Processes known as *producers* add messages to the buffer, while processes known as *consumers* remove them. A producer may insert a message if there is at least one empty slot. A consumer may remove a message if there is at least one full slot. Insertions must be mutually exclusive to preserve the integrity of the buffer, as must deletions be.

```
class BoundedBuffer : private Monitor {
  private:
    int n;
    char *Buf;
    int nextin, nextout, full_cnt;
    Condition notempty, notfull;

  public:
    BoundedBuffer(int size) {
      n = size;
      Buf = new char[n];
      full_cnt = 0;
      nextin = 0;
      nextout = 0;
    }

    void deposit (char data) {
      enter();
      if (full_cnt == N) wait(notfull);
      Buf[nextin] = data;
      nextin += 1;
      full_cnt += 1;
      signal(notempty);
      leave();
    }

    void remove (char &data) {
      enter();
      if (full_cnt == 0) wait(notempty);
      data = Buf[nextout];
      nextout += 1;
      full_cnt -= 1;
      signal(notfull);
      leave();
    }
```

```
};
```

**Discussion** The `remove()` member function should return the `char` data item extracted from the buffer. Because `remove()` must be a `void` function, it uses a reference parameter to return this data. This is a common technique to permit monitor member functions to return values to the calling process.

## 3.7 Correctness

We show the invariance of equations 3.4 through 3.11 by annotating the text of the implementation. Note that these equations need not hold in the middle of atomic actions, but only at the beginning and at the end of such actions. We make the assumption that the monitor has been implemented following the rules outlined in section 3.3.5. This means that `signal()` and `wait()` operations are atomic with respect to each other.

#### Annotated Program for Constructor

```
Monitor::Monitor (void)
{
  busy = FALSE;

  // BEGIN ASSERTIONS
  // ¬busy
  // (3.4) - (3.11)
  // END ASSERTIONS
}
```

#### Annotated Program for Check_Busy

```
atomic void Monitor::check_busy(sync Unary * ptr)
{
  // BEGIN ASSERTIONS
  // (3.4) - (3.11)
  // END ASSERTIONS

  // #became_ready++
```

```
// BEGIN ASSERTIONS
// #enter ≤ #became_ready-1
// busy ∨ (#enter = #became_ready-1)
// (3.4) - (3.10)
// END ASSERTIONS

if (busy == FALSE) {

   // BEGIN ASSERTIONS
   // ¬busy
   // #enter = #became_ready-1
   // #enter = #leave)
   // (3.4) - (3.10)
   // END ASSERTIONS

   busy = TRUE;

   // BEGIN ASSERTIONS
   // busy
   // #enter = #became_ready-1
   // #enter = #leave
   // (3.6) - (3.10)
   // END ASSERTIONS

   *ptr = SET;
   // #enter++

   // BEGIN ASSERTIONS
   // busy
   // (3.4) - (3.11)
   // END ASSERTIONS

}
else {

   // BEGIN ASSERTIONS
   // busy
```

```
    // (3.4) - (3.11)
    // END ASSERTIONS

    Ready.enqueue(ptr);

    // BEGIN ASSERTIONS
    // busy
    // (3.4) - (3.11)
    // END ASSERTIONS

  }

  // BEGIN ASSERTIONS
  // busy
  // (3.4) - (3.11)
  // END ASSERTIONS

}
```

**Annotated Program for Enter**

```
void Monitor::enter(void)
{
  // BEGIN ASSERTIONS
  // (3.4) - (3.11)
  // END ASSERTIONS

  sync Unary *hold = new sync Unary;

  // BEGIN ASSERTIONS
  // (3.4) - (3.11)
  // END ASSERTIONS

  check_busy(hold);

  // BEGIN ASSERTIONS
  // (3.4) - (3.11)
  // END ASSERTIONS
```

```
  if (*hold == SET) delete hold;

  // BEGIN ASSERTIONS
  // (3.4) - (3.11)
  // END ASSERTIONS
}
```

**Annotated Program for Leave**

```
atomic void Monitor::leave(void)
{

  // BEGIN ASSERTIONS
  // busy
  // #enter = #leave+1
  // (3.4) - (3.11)
  // END ASSERTIONS

  // #leave++

  // BEGIN ASSERTIONS
  // busy
  // #leave = #enter
  // (3.6) - (3.11)
  // END ASSERTIONS

  if (Ready.isempty())

    // BEGIN ASSERTIONS
    // busy
    // #leave = #enter
    // #enter = #became_ready
    // (3.6) - (3.11)
    // END ASSERTIONS

    busy = FALSE;

    // BEGIN ASSERTIONS
    // ¬busy
```

```
    // #leave = #became_ready
    // (3.4) - (3.11)
    // END ASSERTIONS

  else

    // BEGIN ASSERTIONS
    // busy
    // #leave = #enter
    // #enter < #became_ready
    // (3.6) - (3.11)
    // END ASSERTIONS

    *(Ready.dequeue()) = SET;
    // #enter++

    // BEGIN ASSERTIONS
    // busy
    // #leave < #became_ready
    // (3.4) - (3.11)
    // END ASSERTIONS

}
```

**Annotated Program for Wait**

```
void Monitor::wait(Condition c)
{

  // BEGIN ASSERTIONS
  // |V^c| = k
  // busy
  // (3.4) - (3.11)
  // END ASSERTIONS

  sync Unary * stop = new sync Unary;

  // BEGIN ASSERTIONS
  // |V^c| = k
```

```
// busy
// (3.4) - (3.11)
// END ASSERTIONS

c.Waiting.enqueue(stop);
// $V_k^c = \mathbf{w}$
// $Q_{k+1}^c = Q_k^c + 1$

// BEGIN ASSERTIONS
// $|V^c| = k + 1$
// busy
// (3.4) - (3.11)
// END ASSERTIONS

leave();

// BEGIN ASSERTIONS
// $|V^c| = k + 1$
// (3.4) - (3.11)
// END ASSERTIONS

if (*stop == SET)

   // BEGIN ASSERTIONS
   // $|V^c| = k + 1$
   // busy
   // (3.4) - (3.11)
   // END ASSERTIONS

   delete stop;

// BEGIN ASSERTIONS
// $|V^c| = k + 1$
// busy
// (3.4) - (3.11)
// END ASSERTIONS

}
```

**Annotated Program for Signal**

**void** `Monitor::signal(Condition &c)`
{

```
// BEGIN ASSERTIONS
```
// $|V^c| = k$
```
// busy
// (3.4) - (3.11)
// END ASSERTIONS
```

**if** `(c.Waiting.isempty() == 0)` {

```
// BEGIN ASSERTIONS
```
// $|V^c| = k$
```
// busy
```
// $Q_k^c > 0$
```
// (3.4) - (3.11)
// END ASSERTIONS

Ready.enqueue(c.Waiting.dequeue());
```
// *#became_ready++*
// $V_k^c = \mathbf{s}_h$
// $Q_{k+1}^c = Q_k^c - 1$

```
// BEGIN ASSERTIONS
```
// $|V^c| = k + 1$
```
// busy
// #enter < #became_ready
// (3.4) - (3.11)
// END ASSERTIONS
```

}
**else** {

```
// BEGIN ASSERTIONS
```
// $|V^c| = k$
```
// busy
```
// $Q_k^c = 0$

```
    // (3.4) - (3.11)
    // END ASSERTIONS

    // V_k^c = s_i
    // Q_{k+1}^c = Q_k^c

    // BEGIN ASSERTIONS
    // |V^c| = k + 1
    // busy
    // Q_{k+1}^c = 0
    // (3.4) - (3.11)
    // END ASSERTIONS

  }

  // BEGIN ASSERTIONS
  // |V^c| = k + 1
  // busy
  // (3.4) - (3.11)
  // END ASSERTIONS

}
```

## 3.8 Proof of Specification

**Proof of (3.1)** :

$$
\begin{array}{ll}
 & \texttt{TRUE} \\
\Leftrightarrow & \quad \{ \text{ law of excluded middle } \} \\
 & \neg\texttt{busy} \vee \texttt{busy} \\
\Leftrightarrow & \quad \{ \text{ by 3.4 and 3.5 } \} \\
 & (\#\texttt{enter} = \#\texttt{leave}) \vee (\#\texttt{enter} = \#\texttt{leave} + 1) \quad \Box
\end{array}
$$

**Proof of (3.2)** :

$$\texttt{TRUE}$$

$$\Leftrightarrow \quad \{ \text{ by 3.11 and 3.4 } \}$$

$$(\#\texttt{enter} = \#\texttt{became\_ready}) \vee (\#\texttt{enter} = \#\texttt{leave} + 1)$$

$$\Leftrightarrow \quad \{ \text{ propery of min, and by 3.1, 3.6 } \}$$

$$(\#\texttt{enter} \geq \min(\#\texttt{became\_ready}, \#\texttt{leave} + 1)) \wedge$$

$$(\#\texttt{enter} \leq \#\texttt{became\_ready}) \wedge (\#\texttt{enter} \leq \#\texttt{leave} + 1)$$

$$\Leftrightarrow \quad \{ \text{ property of min } \}$$

$$(\#\texttt{enter} \geq \min(\#\texttt{became\_ready}, \#\texttt{leave} + 1)) \wedge$$

$$(\#\texttt{enter} \leq \min(\#\texttt{became\_ready}, \#\texttt{leave} + 1))$$

$$\Leftrightarrow \quad \{ \ \}$$

$$\#\texttt{enter} = \min(\#\texttt{became\_ready}, \#\texttt{leave} + 1) \quad \Box$$

**Proof of (3.3)** : To show that $V^c$ is a prefix of a string in the language generated by the context-free grammar given in the specification, we must show the validity of the following three implications:

1. $(|V^c_{0..k-1} \restriction \{\mathbf{s}_h\}| < |V^c_{0..k-1} \restriction \{\mathbf{w}\}|) \wedge (|V^c| \geq k+1) \Rightarrow V^c_k \in \{\mathbf{w}, \mathbf{s}_h\}$
2. $(|V^c_{0..k-1} \restriction \{\mathbf{s}_h\}| = |V^c_{0..k-1} \restriction \{\mathbf{w}\}|) \wedge (|V^c| \geq k+1) \Rightarrow V^c_k \in \{\mathbf{w}, \mathbf{s}_i\}$
3. $|V^c_{0..k-1} \restriction \{\mathbf{s}_h\}| > |V^c_{0..k-1} \restriction \{\mathbf{w}\}| \Rightarrow \texttt{FALSE}$

We prove each of these in turn.

1.

$$(|V^c_{0..k-1} \restriction \{\mathbf{s}_h\}| < |V^c_{0..k-1} \restriction \{\mathbf{w}\}|) \wedge (|V^c| \geq k+1)$$

$$\Leftrightarrow \quad \{ \text{ by 3.8 } \}$$

$$(Q^c_k \geq 0) \wedge (|V^c| \geq k+1)$$

$$\Rightarrow \quad \{ \text{ by 3.10 } \}$$

$$V^c_k \neq \mathbf{s}_i$$

$$\Leftrightarrow \quad \{ \text{ since } V^c_{k+1} \in \Sigma = \{\mathbf{w}, \mathbf{s}_h, \mathbf{s}_i\} \ \}$$

$$V^c_k \in \{\mathbf{w}, \mathbf{s}_h\}$$

2.

$$(|V^c_{0..k-1} \restriction \{\mathbf{s}_h\}| = |V^c_{0..k-1} \restriction \{\mathbf{w}\}|) \wedge (|V^c| \geq k+1)$$

$$\Leftrightarrow \quad \{ \text{ by 3.8 } \}$$

$$
\begin{array}{ll}
 & (Q_k^c = 0) \wedge (|V^c| \geq k+1) \\
\Rightarrow & \quad \{ \text{ by 3.9 } \} \\
 & V_k^c \neq \mathbf{s}_h \\
\Leftrightarrow & \quad \{ \text{ since } V_{k+1}^c \in \Sigma = \{\mathbf{w}, \mathbf{s}_h, \mathbf{s}_i\} \ \} \\
 & V_k^c \in \{\mathbf{w}, \mathbf{s}_i\}
\end{array}
$$

3.
$$
\begin{array}{ll}
 & |V_{0..k-1}^c \restriction \{\mathbf{s}_h\}| > |V_{0..k-1}^c \restriction \{\mathbf{w}\}| \\
\Leftrightarrow & \quad \{ \text{ by 3.8 } \} \\
 & Q_k^c < 0 \\
\Leftrightarrow & \quad \{ \text{ by 3.7 } \} \\
 & \texttt{FALSE} \quad \square
\end{array}
$$

## 3.9 Interpretation of Grammar

**Lemma** : Given the context-free grammar $G$ defined by start symbol $S$, the set $\Sigma = \{\#, [, ]\}$ of terminals, and the production rules:

$$
\begin{array}{lcl}
S & \rightarrow & \#S \mid BS \mid \epsilon \\
B & \rightarrow & B[B] \mid \epsilon
\end{array}
$$

Let $P$ denote the set

$$\{(\#^* b \#^*)^* \mid b \text{ is a string of properly balanced square brackets}\}$$

and let $\mathcal{L}(G)$ denote the context-free language corresponding to $G$.

Then we have:

$$\mathcal{L}(G) = P$$

**Proof** : We show equality of the two sets by showing that $w \in \mathcal{L}(G) \Leftrightarrow w \in P$.

a) We first show $w \in \mathcal{L}(G) \Rightarrow w \in P$ by induction on $|w|$.

*Base case.* For $|w| = 0$, clearly $w = \epsilon$ and so is in $P$ trivially. For $|w| = 1$, there is only one production rule which generates a single terminal symbol (*i.e.* $S \rightarrow \#S$) and therefore $w = \#$. Again, clearly $w \in P$.

*Inductive hypothesis.* Suppose that for all $w \in \mathcal{L}(G)$ such that $|w| = k - 2$, $w \in P$.

*Inductive step.* Let $w$ be a string in $\mathcal{L}(G)$ such that $|w| = k$. If $w$ contains no square brackets, clearly $w \in P$. If $w$ does contain square brackets, it must contain an equal number of each (since the only production rule for square brackets, $B \rightarrow B[B]$, creates one of each). Furthermore, there must be at least one [ symbol followed immediately by a ] symbol. Thus, $\exists x, y : x, y \in \Sigma^* : w = x[]y$. Now note that this pair of square brackets must have been generated by an application of the production rule $B \rightarrow B[B]$ followed by an application of $B \rightarrow \epsilon$ applied to the second $B$. That is, $B \rightarrow B[B] \rightarrow B[]$. Hence, by omitting these two productions in the derivation of $w$, the string $xy$ is obtained. Thus, $xy \in \mathcal{L}(G)$. But $|xy| = k - 2$, and so $xy \in P$. Hence, $w \in P$.

b) We now show $w \in P \Rightarrow w \in \mathcal{L}(G)$, again by induction on $|w|$.

*Base case.* For $|w| = 0$, clearly $w = \epsilon$ and so is in $\mathcal{L}(G)$ (since it can be generated by $S \rightarrow \epsilon$). For $|w| = 1$, we see that $w = \#$. Again, $w \in \mathcal{L}(G)$ since it is generated by $S \rightarrow \#S \rightarrow \#$.

*Inductive hypothesis.* Suppose that for all $w \in P$ such that $|w| = k - 2$, $w \in \mathcal{L}(G)$.

*Inductive step.* Let $w$ be a string in $P$ such that $|w| = k$. If $w = \#^k$, it can be generated by $k$ applications of the rule $S \rightarrow \#S$ followed by one application of $S \rightarrow \epsilon$. If $w$ does contain square brackets, it must contain at least one [ symbol which is immediately followed by a ] symbol (since the square brackets are properly balanced). Thus, $\exists x, y : x, y \in \Sigma^* : w = x[]y$. Now, since $w \in P$, certainly $xy \in P$ (by a property of nested square brackets) and also $|xy| = k - 2$, so $xy \in \mathcal{L}(G)$. It can be easily shown that any derivation of $xy$ in $G$ can be extended to form a derivation of $x[]y$. Hence, $w \in \mathcal{L}(G)$.

Thus, $\mathcal{L}(G) = P$. $\square$

## 3.10 Confidence in Specification

How can we be confident that the specification outlined in section 3.2 is consistent with the accepted definition of a monitor? The specification of a monitor is often given operationally. An informal description of the behaviour of a monitor is usually followed by a collection of proof rules or an implementation using semaphores. For our purposes, the former is insufficient and the latter unsatisfying.

A collection of axiomatic proof rules is not sufficient because they often neglect to enforce any kind of progress condition. The proof rules introduced in Hoare's classic paper [9] allow a valid implementation of `wait` to be "exit the monitor and suspend forever". Other collections of proof rules ([11] and [2]) have similar flaws, allowing spurious suspensions. It is a general observation that the specification of a monitor implies the proof rules, but not the converse.

When a monitor is defined by its implementation using semaphores, the entire specification, including progress properties, is certainly captured with the specification of semaphores. This, however, ties the specification to a particular implementation and it is unclear how to cleanly prove that a given implementation of a monitor meets the required safety, progress, and fairness conditions.

To support the claim that the properties given in section 3.2 are a complete and consistent definition of a monitor, we show their equivalence to a state transition diagram. The assumption being made is that the state transition diagram is conceptually closer to the popular definition of a monitor.

**Operational Specification.** As described in section 3.4.1, the state of a monitor is given by the collection of processes executing inside (*in*), the queue of ready processes (*rq*), and the queues of waiting processes associated with each condition variable (we simplify the exposition by considering only one condition variable, and hence only one associated queue, *cq*). We use the tuple (*in*, *rq*, *cq*) to denote the corresponding state. The symbol $\phi$ is used to denote the empty set, and lower case Greek letters ($\rho$ and $\pi$) denote individual processes. The symbols $+$ and $-$ are used as superscripts to represent addition and removal from a queue. For example, $\mathrm{q}^{+\rho}$ denotes the enqueuing of the process $\rho$ to q, and $\mathrm{q}^{-\pi}$ denotes the dequeuing of a process from q and assigns the symbol $\pi$ to represent it. The following, therefore, is an exhaustive list of the valid transitions for a monitor:

call:

$$(\{\rho\}, rq, cq) \quad \longmapsto \quad (\{\rho\}, rq^{+\pi}, cq) \tag{3.12}$$

$$(\phi, \phi, cq) \quad \longmapsto \quad (\{\rho\}, \phi, cq) \tag{3.13}$$

depart:

$$\begin{aligned} (\{\rho\}, \phi, cq) &\longmapsto (\phi, \phi, cq) & (3.14)\\ (\{\rho\}, rq \neq \phi, cq) &\longmapsto (\{\pi\}, rq^{-\pi}, cq) & (3.15)\end{aligned}$$

signal:

$$\begin{aligned} (\{\rho\}, rq, \phi) &\longmapsto (\{\rho\}, rq, \phi) & (3.16)\\ (\{\rho\}, rq, cq \neq \phi) &\longmapsto (\{\rho\}, rq^{+\pi}, cq^{-\pi}) & (3.17)\end{aligned}$$

wait:

$$\begin{aligned} (\{\rho\}, \phi, cq) &\longmapsto (\phi, \phi, cq^{+\rho}) & (3.18)\\ (\{\rho\}, rq \neq \phi, cq) &\longmapsto (\{\pi\}, rq^{-\pi}, cq^{+\rho}) & (3.19)\end{aligned}$$

The initial state of a monitor is $(\phi, \phi, \phi)$.

**Axiomatic Specification.** Note that a state transition can be characterized by the movement of processes between the four collections associated with a monitor: processes that are outside (*out*), those that are executing inside (*in*), those that are ready to execute (*rq*), and those that are waiting on a condition variable (*cq*). We use the notation $\langle i \rightarrow j\rangle$ to denote a transition in which a process is moved from collection $i$ to collection $j$. Thus, the four functions on monitors as defined in section 3.2 can be described as follows:

$$\begin{aligned} \text{call} &: \langle out \rightarrow in\rangle \vee \langle out \rightarrow rq\rangle & (3.20)\\ \text{depart} &: \langle in \rightarrow out\rangle \vee (\langle in \rightarrow out\rangle ; \langle rq \rightarrow in\rangle) & (3.21)\\ \text{signal} &: \text{noop} \vee \langle cq \rightarrow rq\rangle & (3.22)\\ \text{wait} &: \langle in \rightarrow cq\rangle \vee (\langle in \rightarrow cq\rangle ; \langle rq \rightarrow in\rangle) & (3.23)\end{aligned}$$

Now we introduce the notation $[i^+]$ to mean the number of transitions that have increased the size of collection $i$, and $[i^-]$ to mean the number that have decrease the size of collection $i$. Thus, the definitions of the ghost variables introduced in section 3.2 can be written:

$$\begin{aligned} \#\texttt{enter} &= [in^+] + [rq^-] & (3.24)\\ \#\texttt{leave} &= [in^-] + [rq^-] & (3.25)\\ \#\texttt{became\_ready} &= [in^+] + [rq^+] & (3.26)\end{aligned}$$

$$
\begin{aligned}
V_k^c = \mathbf{s}_i &\Leftrightarrow \text{ noop is } k^{th} \text{ transition on } c && (3.27)\\
V_k^c = \mathbf{s}_h &\Leftrightarrow \langle cq \rightarrow rq\rangle \text{ is } k^{th} \text{ transition on } c && (3.28)\\
V_k^c = \mathbf{w} &\Leftrightarrow (\langle in \rightarrow cq\rangle \text{ is } k^{th} \text{ transition on } c) \vee \\
&\quad\ (\langle in \rightarrow cq\rangle;\langle rq \rightarrow in\rangle \text{ is } k^{th} \text{ transition on } c) && (3.29)
\end{aligned}
$$

**Claim** : The specifications (3.1) - (3.3) are consistent with the operational specification of a monitor given by (3.12) - (3.19)

**Proof** : Let #i denote the number of transitions labelled i which have occurred. We begin by noting the following inequalities:

$$
\begin{aligned}
\#3.13 &\leq \#3.14 + \#3.18 + 1 && (3.30)\\
\#3.14 + \#3.18 &\leq \#3.13 && (3.31)\\
\#3.17 &\leq \#3.18 + \#3.19 && (3.32)\\
\#3.15 + \#3.19 &\leq \#3.12 + \#3.17 && (3.33)
\end{aligned}
$$

Also, by the definition of these terms (in (3.24) -(3.29)) we have:

$$
\begin{aligned}
\#\texttt{enter} &= \#3.13 + \#3.15 + \#3.19\\
\#\texttt{leave} &= \#3.14 + \#3.15 + \#3.18 + \#3.19\\
\#\texttt{became\_ready} &= \#3.12 + \#3.13 + \#3.17\\
V_k^c = \mathbf{s}_i &\Leftrightarrow 3.16 \text{ is } k^{th} \text{ transition on } c\\
V_k^c = \mathbf{s}_h &\Leftrightarrow 3.17 \text{ is } k^{th} \text{ transition on } c\\
V_k^c = \mathbf{w} &\Leftrightarrow 3.18 \text{ or } 3.19 \text{ is } k^{th} \text{ transition on } c
\end{aligned}
$$

We now show that each of the specifications in 3.2 is implied by the operational description given by 3.12-3.19.

**For (3.1):**

$$
\begin{aligned}
&\texttt{TRUE}\\
\Leftrightarrow &\quad \{ \text{ by 3.30 and 3.31 and } \#3.13 \in \mathbb{W} \ \}\\
&(\#3.13 = \#3.14 + \#3.18) \vee \#3.13 = \#3.14 + \#3.18 + 1)\\
\Leftrightarrow &\quad \{ \text{ by definition of } \texttt{\#enter} \text{ and } \texttt{\#leave} \ \}\\
&(\#\texttt{enter} = \#\texttt{leave}) \vee \#\texttt{enter} = \#\texttt{leave} + 1)
\end{aligned}
$$

**For (3.2):**

$$
\begin{array}{ll}
& \texttt{TRUE} \\
\Leftrightarrow & \quad \{ \text{ by } 3.33 \text{ } \} \\
& (\#3.12 + \#3.17 > \#3.19 + \#3.15) \vee (\#3.12 + \#3.17 = \#3.19 + \#3.15) \\
\Leftrightarrow & \quad \{ \text{ since initial state is } (\phi, \phi, \phi) \text{ } \} \\
& (\mathit{state} \neq (\phi, \phi, cq)) \vee (\#3.12 + \#3.17 = \#3.19 + \#3.15) \\
\Leftrightarrow & \quad \{ \text{ since initial state is } (\phi, \phi, \phi) \text{ } \} \\
& (\#3.13 \neq \#3.14 + \#3.18) \vee (\#3.12 + \#3.17 = \#3.19 + \#3.15) \\
\Leftrightarrow & \quad \{ \text{ by } 3.30 \text{ and } 3.31 \text{ } \} \\
& (\#3.13 = \#3.14 + \#3.18 + 1) \vee (\#3.12 + \#3.17 = \#3.19 + \#3.15) \\
\Leftrightarrow & \quad \{ \text{ by } 3.30 \text{ and } 3.33 \text{ } \} \\
& ((\#3.13 = \#3.14 + \#3.18 + 1) \vee (\#3.12 + \#3.17 = \#3.19 + \#3.15)) \\
& \wedge (\#3.13 \leq \#3.14 + \#3.18 + 1) \wedge (\#3.15 + \#3.19 \leq \#3.12 + \#3.17) \\
\Leftrightarrow & \quad \{ \text{ property of min } \} \\
& \#3.13 + \#3.15 + \#3.19 = \\
& \min(\#3.12 + \#3.13 + \#3.17, \#3.14 + \#3.15 + \#3.18 + \#3.19 + 1) \\
\Leftrightarrow & \quad \{ \text{ by definition of } \texttt{\#enter} \text{ and } \texttt{\#leave} \text{ and } \texttt{\#became\_ready} \text{ } \} \\
& \#\texttt{enter} = \min(\#\texttt{became\_ready}, \#\texttt{leave} + 1)
\end{array}
$$

**For (3.3):** We must show the following:

1. $\#3.18 + \#3.19 > \#3.17 \Rightarrow 3.16$ cannot be the next transition
2. $\#3.18 + \#3.19 = \#3.17 \Rightarrow 3.17$ cannot be the next transition
3. $\#3.18 + \#3.19 = \#3.17 \Rightarrow \texttt{FALSE}$

We show each of these in turn.

1. $$
\begin{array}{ll}
& \#3.18 + \#3.19 > \#3.17 \\
\Leftrightarrow & \quad \{ \text{ definition of transitions 3.17-3.19 } \}
\end{array}
$$

$cq \neq \phi$

$\Rightarrow$ { definition of transition 3.16 }

3.16 cannot be the next transition

2. $\#3.18 + \#3.19 = \#3.17$

$\Leftrightarrow$ { definition of transitions 3.17-3.19 and $cq = \phi$ in initial state }

$cq = \phi$

$\Rightarrow$ { definition of transition 3.17 }

3.17 cannot be the next transition

3. $\#3.18 + \#3.19 < \#3.17$

$\Leftrightarrow$ { by 3.32 }

`FALSE`

Thus, we have established that the specification in section 3.2 is consistent with the operational specification given by transitions (3.12)-(3.19) and initial state $(\phi, \phi, \phi)$. $\square$

We now establish the converse of the above result. Namely, that the operational specification given by transitions (3.12)-(3.19) and initial state $(\phi, \phi, \phi)$ is consistent with the specification in section 3.2. That is, the specification in section 3.2 implies the operation specification. We proceed by first showing that both specifications subtend the same valid state space. We then establish that each transition given in the operational specification is a consequence of the valid transitions outlined in section 3.2 (and redescribed in (3.20) - (3.23)) and the constraint imposed by the valid state space.

**Lemma 1** : Let $S$ denote the set

$$\{(in, rq, cq) \mid (|in| = 1) \vee (|in| = 0 \wedge |rq| = 0)\}$$

Then the set of valid states of a monitor (as defined operationally by (3.12)-(3.19)) is $S$.

**Proof of Lemma 1** :

The set $S$ is represented pictorially in Figure 3.2, where $\#$`Inside` $= |in|$, $\#$`Ready` $= |rq|$, and $\#$`Waiting` $= |cq|$.

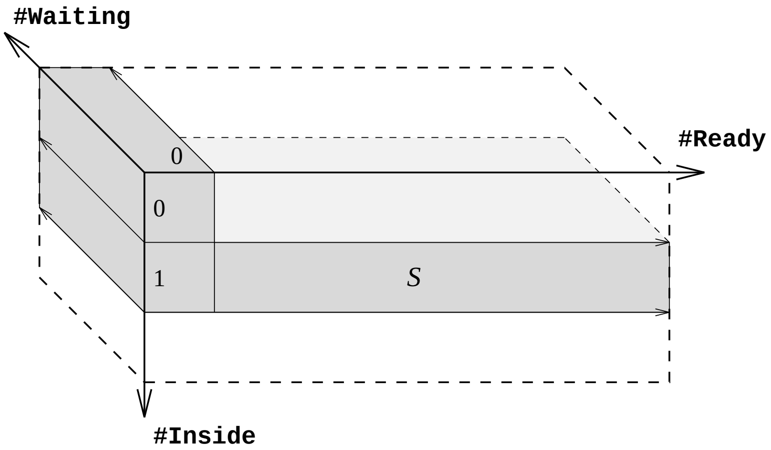

Figure 3.2: Valid States for a Monitor

Clearly the initial state $(\phi, \phi, \phi)$ is in $S$. Also, it is easy to verify that $S$ is closed under the transitions (3.12)-(3.19). Thus, the set of valid states of a monitor is a subset of $S$.

Now we verify that every state in $S$ can be reached from $(\phi, \phi, \phi)$ using the transitions (3.12)-(3.19). First, a state $(\phi, \phi, cq)$ where $|cq| = k$ can be reached by transition 3.13 followed by transition 3.17, all repeated $k$ times (*i.e.* $(3.13; 3.17)^k$ ). Also, a state $(\{\rho\}, rq, cq)$ where $|rq| = k$, $|cq| = l$ can be reached by transition 3.13, followed by $k + l$ transition 3.12's, followed by $l$ transition 3.19's (*i.e.* $3.13; 3.12^{k+l}; 3.19^l$). Thus, $S$ is a subset of the valid states of a monitor.

Hence, $S$ is the set of valid states of a monitor (as defined operationally by (3.12)-(3.19)). $\square$

**Lemma 2** : Let $S$ denote the set

$$\{(in, rq, cq) \mid (|in| = 1) \vee (|in| = 0 \wedge |rq| = 0)\}$$

Then the set of valid states of a monitor (as defined axiomatically in section 3.2)) is $S$.

**Proof of Lemma 2** :

TRUE

$\Leftrightarrow$ { by 3.2 and definitions 3.24-3.26 }

$[in^+] + [rq^-] = \min([in^+] + [rq^+], [in^-] + [rq^-] + 1)$

$\Leftrightarrow$ { property of min }

$([rq^-] = [rq^+]) \vee ([in^+] = [in^-] + 1)$

$\Leftrightarrow$ { by 3.1 and definitions 3.24, 3.25 }

$(([rq^+] = [rq^-]) \vee ([in^+] = [in^-] + 1)) \wedge$
$(([in^+] + [rq^-] = [in^-] + [rq^-]) \vee ([in^+] + [rq^-] = [in^-] + [rq^-] + 1))$

$\Leftrightarrow$ { property of $\leq$ }

$(([rq^+] = [rq^-]) \vee ([in^+] = [in^-] + 1)) \wedge$
$(([in^+] = [in^-]) \vee ([in^+] = [in^-] + 1))$

$\Leftrightarrow$ { DeMorgan's Laws }

$([in^+] = [in^-] + 1) \vee (([in^+] = [in^-]) \wedge ([rq^+] = [rq^-]))$

Since the initial state has $|in| = |rq| = |cq| = 0$, therefore the set of states permitted by constraints (3.1) and (3.2) is precisely the set $S$. $\square$

**Claim** : The operational specification (3.12) - (3.19) is consistent with the specification of a monitor given in section 3.2.

**Proof** : We begin by noting that both specifications yield the same set of valid states (from Lemmas 1 and 2). Let $S$ denote this set.

We can now show that each transition given in the operational specification corresponds to one of the operations defined in section 3.2 (and redescribed by (3.20) - (3.23), and pictorially represented in Figure 3.3) subject to the constraint imposed by $S$.

For example, transitions (3.12) and (3.13) stem from the definition of call in (3.20). If there are no processes inside the monitor, the $\langle out \rightarrow rq \rangle$ transition is disallowed by $S$. Thus, a call in this case must be a $\langle out \rightarrow in \rangle$ transition, which is precisely transition (3.13). Conversely, if there is a process inside the monitor, then the $\langle out \rightarrow in \rangle$ transition is disallowed. In this case a call must be a $\langle out \rightarrow rq \rangle$ transition, which is precisely transition (3.12).

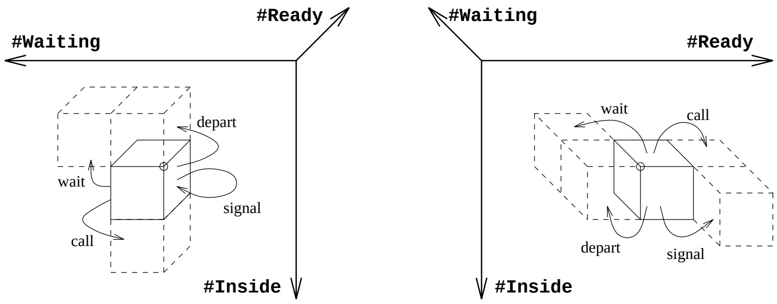


Figure 3.3: Valid State Transitions for a Monitor

A similar argument can be used to show how the depart transitions ((3.14) and (3.15)) are implied by (3.21), and how the wait transitions ((3.18) and (3.19)) are implied by (3.23).

This argument does not, however, suffice for the signal transitions, since (3.22) permits a noop as a valid transition, which will never violate any constraint imposed by $S$. For these signal transitions ((3.16) and (3.17)), specification (3.3) is required. Equation (3.3) tells us that if a signal is performed and there are no waiting processes, the signal is ignored. This is precisely transition (3.16). Equation (3.3) also tells us that if a signal is performed and there are waiting processes, the signal is heard. From the definition of a heard signal (*i.e.* $\langle cq \rightarrow rq \rangle$), we see that this is precisely transition (3.17).

Thus, the operational specification given by (3.12) - (3.19) is consistent with the specification given in section 3.2. $\square$

# Chapter 4

# Asynchronous Channels

## 4.1 Introduction

An asynchronous channel is a first-in-first-out message-passing buffer. Two operations are defined on such a channel:

- The `nonblockingSend()` operation places a message in the buffer. It never suspends. Processes that send messages are called *producers*.
- The `blockingReceive()` operation removes the next message from the buffer. If there is no message to be removed, this operation suspends until there is such a message. Processes that receive messages are called *consumers*.

The channel can be used by arbitrary and varying numbers of producers and consumers. Asynchronous channels are discussed in [2] and [5].

## 4.2 Specification

Since the `blockingReceive()` operation can suspend, it is not atomic, and we distinguish between the initiation and the termination of this operation. Let `iR` be the number of `blockingReceive()` operations that have been *initiated*, and let `cR` be the number of `blockingReceive()` operations that have *completed*. Since the `nonblockingSend()` operation is atomic, no such distinction is required, and we define `cS` to be the number of completed `nonblockingSend()` operations.

The first safety condition is that as many `blockingReceive()` operations as possible have completed, subject to the constraints of the number of initiated `blockingReceive()` operations and the number of completed `nonblockingSend()` operations.

$$\mathtt{cR} = \min(\mathtt{iR}, \mathtt{cS}) \tag{4.1}$$

In addition, another safety condition is that the channel delivers the messages in FIFO order. Let $\mathtt{s}_{(k)}$ be the message sent by the $(k+1)^{th}$ `nonblockingSend` executed on the channel, for $0 \leq k < \mathtt{cS}$. Let $\mathtt{r}_{(k)}$ be the message received by the $(k+1)^{th}$ `blockingReceive()` executed on the channel, for $0 \leq k < \mathtt{cR}$. The $(k+1)^{th}$ message received is the $(k+1)^{th}$ message sent.

$$\forall k : 0 \leq k < \mathtt{cR} : \mathtt{r}_{(k)} = \mathtt{s}_{(k)} \tag{4.2}$$

## 4.3 The Design

### 4.3.1 Asynchronous Channel as a Class

An asynchronous channel is implemented as a CC++ class. It stores the sent messages that have not yet been received in a private queue, `Undelivered`. The class has two public member functions: `nonblockingSend()` and `blockingReceive()`. Since the `nonblockingSend()` member function never suspends, and since it manipulates the `Undelivered` queue, this function is atomic. The member function `blockingReceive()`, however, can suspend, and so is not atomic. The class also contains a queue, `EmptySlots`, to keep track of the suspended `blockingReceive()` operations.

The channel implementation should be general enough to permit any object, including user-defined structures, to be passed as a message. This generality is achieved (without indirection and without sacrificing the benefits of type-checking) by making use of templates.

### 4.3.2 Messages and Slots

Producers send messages on the channel and consumers give empty *slots* to the channel, as shown in Figure 4.1. A slot can contain at most one message. The only transition that is possible in the state of a slot is the transition from empty to full. A consumer gives an empty slot to the channel, waits for the slot to become full, and then processes the message in the slot.


Figure 4.1: Channel Overview

Slots are implemented as `sync` pointers. An empty slot is an undefined `sync` pointer. A full slot is a `sync` pointer that has been defined. The pointer points to the message contained in the slot.

A `blockingReceive()` operation begins by creating a slot for a message. If there is an undelivered message, the slot is filled and the operation terminates. If there is no such message, the slot is added to a queue of such slots, the `EmptySlots` queue. The `blockingReceive()` operation then suspends on the contents of this slot (see Figure 4.2).

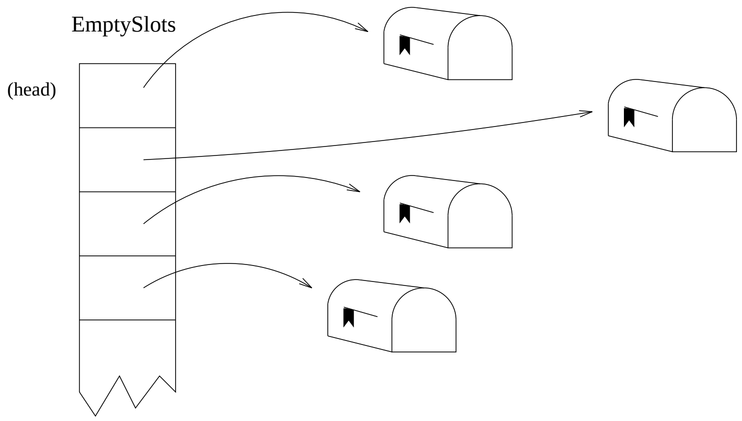


Figure 4.2: Queue of Empty Slots

A `nonblockingSend()` operation begins by making a copy of the message being sent. If there are any suspended `blockingReceive()` operations, the first element of `EmptySlots` is dequeued and the `sync` pointer of the slot is defined to point to the message copy. If there are none, the message copy is appended to a queue of undelivered messages, `Undelivered` (see Figure 4.3).

This copying of the message being sent has two effects. First, the producer is free to discard the message once the `nonblockingSend()` operation has terminated. Second, the consumer is responsible for deallocating the memory for the message that is received.

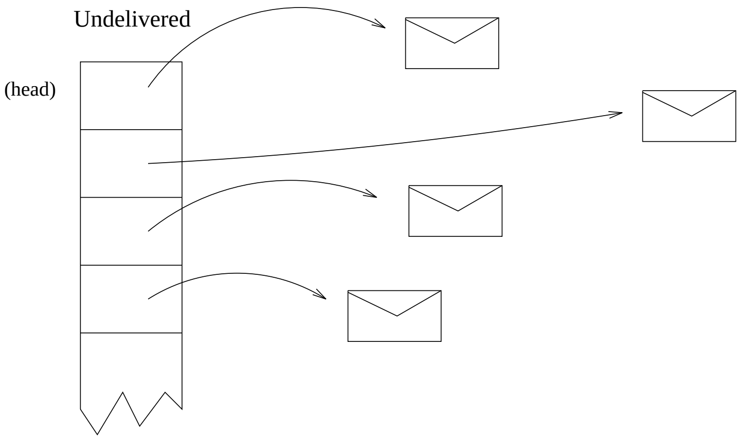


Figure 4.3: Queue of Undelivered Messages

## 4.4 Implementation State and Properties

### 4.4.1 State

The state of an asynchronous channel is given by:

1. A queue of empty slots called `EmptySlots`. The number of elements in this queue, `#EmptySlots`, is the number of `blockingReceive()` operations that have been initiated, but have not yet completed.

$$\#\texttt{EmptySlots} = \texttt{iR} - \texttt{cR} \tag{4.3}$$

   Empty slots are placed in the queue `EmptySlots` in order.

$$\forall j : 0 \leq j < \#\texttt{EmptySlots} : \texttt{EmptySlots}[j] = \texttt{r}_{(j+\texttt{cR})} \tag{4.4}$$

   where $\texttt{EmptySlots}[j]$ is the $(j+1)^{th}$ element of `EmptySlots`.

2. A queue of undelivered messages called `Undelivered`. Let the number of elements in this queue be `#Undelivered`. Since each completed `nonblockingSend()` either adds a message to this queue or allows a `blockingReceive()` operation to complete, we have:

$$\#\texttt{Undelivered} = \texttt{cS} - \texttt{cR} \tag{4.5}$$

Undelivered messages are placed in the queue Undelivered in order.

$$\forall j : 0 \leq j < \#\texttt{Undelivered} : \texttt{Undelivered}[j] = \texttt{s}_{(j+\texttt{cR})} \qquad (4.6)$$

where Undelivered$[j]$ is the $(j+1)^{th}$ element of Undelivered.

### 4.4.2 Properties

**Boundedness Requirement.** The number of blockingReceive() operations that can complete is bounded by the number of nonblockingSend() operations that have completed.

$$\texttt{cR} \leq \texttt{cS} \qquad (4.7)$$

**The Set of Suspended Processes is Minimal.** A process can be suspended only if its completion would violate the safety condition given by (4.1).

$$(\#\texttt{EmptySlots} = 0) \vee (\#\texttt{Undelivered} = 0) \qquad (4.8)$$

**Ordering Requirement.** The value returned by the $(k+1)^{th}$ blockingReceive() points to the message contained in the $(k+1)^{th}$ slot. Let $\texttt{slot}_{(k)}$ be the message contained in the $(k+1)^{th}$ slot.

$$\forall k : 0 \leq k < \texttt{cR} : \texttt{r}_{(k)} = \texttt{slot}_{(k)} \qquad (4.9)$$

Also, the $(k+1)^{th}$ message sent is put in the $(k+1)^{th}$ slot.

$$\forall k : 0 \leq k < \texttt{cR} : \texttt{s}_{(k)} = \texttt{slot}_{(k)} \qquad (4.10)$$

## 4.5 Implementation

```
template <class Message>
class AChannel {
  private:
    Queue <Message * sync> EmptySlots;
    Queue <Message> Undelivered;

    atomic void give_slot (Message * sync *);
```

```
  public:
    atomic void nonblockingSend (Message &);
    Message * blockingReceive(void);
};
```

**Give_Slot**

```
template <class Message>
atomic void AChannel<Message>::give_slot (Message * sync * p)
{
  if (Undelivered.isempty()) EmptySlots.append(p);
  else p = Undelivered.dequeue();
}
```

**Nonblocking Send**

```
template <class Message>
atomic void AChannel<Message>::nonblockingSend (Message &M) {
  Message * q = new Message (M);
  if (EmptySlots.isempty()) Undelivered.append(q);
  else EmptySlots.dequeue() = q;
}
```

**Blocking Receive**

```
template <class Message>
Message * AChannel<Message>::blockingReceive (void) {
  Message * sync * p = new Message * sync;
  give_slot(p);
  return p;                        //suspends here if slot is empty
}
```

## 4.6 Examples

### 4.6.1 Producer/Consumer

**Motivation** Our first example is a simple illustration of how to instantiate and use a basic asynchronous channel.

**Problem Description** Two processes, a producer and a consumer, communicate over a channel. The producer creates data which the consumer then uses.

```
AChannel<float> C;          //declaration of basic asynchronous channel
const int N = 10;                                  //number of iterations

void Producer (void) {
  float out_data;
  for (int i=0; i<N; i++) {
    produce (&out_data);
    C.nonblockingSend(out_data);
  }
}

void Consumer (void) {
  float * in_data;
  for (int i=0; i<N; i++) {
    in_data = C.blockingReceive();
    consume (*in_data);
    delete in_data;
  }
}

main()
{
  par {
    Producer();
    Consumer();
  }
}
```

**Discussion** In this example, a channel of `float`s is created using the usual C++ template instantiation mechanism. Notice that the consumer receives a *pointer* to a `float` value, and that it is up to the consumer to deallocate this memory (with `delete`).

### 4.6.2 Dirichlet's Problem

**Motivation** Arrays of channels can be declared and used. Channels should be passed to functions as reference parameters.

**Problem Description** Each process in a mesh begins with an initial value. At each iteration, every process calculates its new value as a weighted average of its old value and the values of its neighbours. The problem is to find the final values to which the processes converge.

```
const int size = 7;           //size of mesh (no. of cells on a side)
const int N = 100;                                //number of iterations

typedef float VType;        //type of values stored and communicated

VType Cell (AChannel<VType> &f_left, AChannel<VType> &f_right,
            AChannel<VType> &f_above, AChannel<VType> &f_below,
            AChannel<VType> &t_left, AChannel<VType> &t_right,
            AChannel<VType> &t_above, AChannel<VType> &t_below,
            VType value) {
  VType *left_val, *right_val, *above_val, *below_val;
  for (int time=0; time<N; time++) {
    par {
      t_left.nonblockingSend(value);
      t_right.nonblockingSend(value);
      t_above.nonblockingSend(value);
      t_below.nonblockingSend(value);
      left_val = f_left.blockingReceive();
      right_val = f_right.blockingReceive();
      above_val = f_above.blockingReceive();
      below_val = f_below.blockingReceive();
    }
    value = (4*value + *left_val + *right_val +
             *above_val + *below_val )/8.0;
    par {
      delete left_val;
      delete right_val;
      delete above_val;
      delete below_val;
    }
  }
  return value;
}

VType Boundary (AChannel<VType> &f_int, AChannel<VType> &t_int,

                const VType value) {
  VType * in_val;
  parfor (int time=0; time<N; time++) {
    par {
      t_int.nonblockingSend(value);
      in_val = f_int.blockingReceive();
      delete in_val;
    }
  }
```

```
  return value;
}

void
main ()
{
  AChannel<VType> to_right[size][size], to_left[size][size],
               to_above[size][size], to_below[size][size];
    //to_x[i][j] denotes Cell ij's output channel in direction x

  parfor (int i=1; i<size-1; i++) {
    par {
      Boundary(to_above[1][i],to_below[0][i], i);
      Boundary(to_right[i][size-2],to_left[i][size-1],i+size);
      Boundary(to_below[size-2][i],to_above[size-1][i],i+size*2);
      Boundary(to_left[i][1],to_right[i][0],i+size*3);
      parfor (int j=1; j<size-1; j++)
        Cell(to_right[i][j-1], to_left[i][j+1],
             to_below[i-1][j], to_above[i+1][j],
             to_left[i][j], to_right[i][j],
             to_above[i][j], to_below[i][j],0);
    }  /*par*/
  }  /*parfor*/

}
```

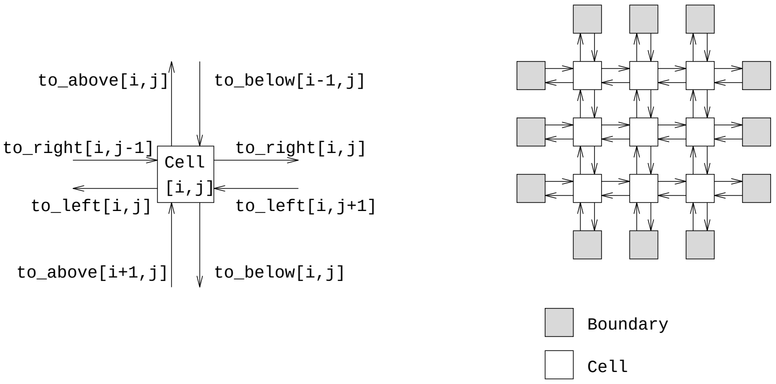


Figure 4.4: Mesh Structure for Dirichlet's Problem

**Discussion** A mesh of processes communicating via channels can also be set up by recursion, as is done in [6, Section 8.2]. This avoids the declaration of a global array of channels. An iterative approach was chosen here for the sake of clarity of code and to illustrate the integration of channels and arrays.

Also note that the channels are passed as reference parameters. Because of the pass-by-value semantics of C++, if a channel is not passed by reference, a local copy of the channel is made. Sends and receives performed on this local copy do not affect the original channel.

### 4.6.3 Mutual Exclusion with a Token Ring

**Motivation** User-defined structures can be sent over channels. Channels can have multiple writers.

**Problem Description** We have a collection of processes that require mutually exclusive access to a critical section. Each process is decomposed into a server and a client process. The servers are arranged in a ring. A token is used to control access to the critical section. The token is passed clockwise around the ring and requests for the token are passed counterclockwise.

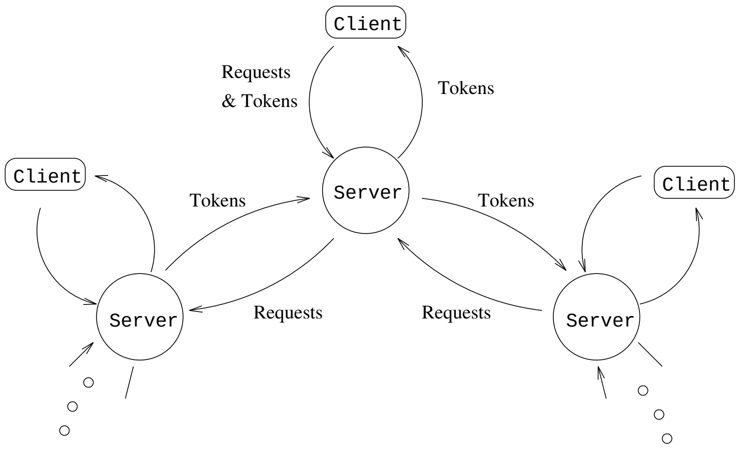


Figure 4.5: Token Ring of Processes for Mutual Exclusion to a Critical Section

```
const int size = 10;                //number of servers in token ring
const int N = 20;                         //number of iterations

enum MType {TOKEN, C_REQUEST, S_REQUEST};
```

```
void Client (int id, AChannel<MType> *In_Msgs,
             AChannel<MType> *Out_Msgs) {
  for (int i=0; i<N; i++) {
    /*non-critical section*/
    Out_Msgs→nonblockingSend(C_REQUEST);      //request to enter cs
    In_Msgs→blockingReceive();           //receive token from server
    /*critical section*/
    Out_Msgs→nonblockingSend(TOKEN);   //return token at end of cs
  }
}

void Server (int id,
       AChannel<MType> *In_Msgs, AChannel<MType> *Prev_Out,
       AChannel<MType> *Next_Out, AChannel<MType> *Clt_Out) {
  //definition omitted for brevity
}

void create_ring_global (void)
{
  //form a ring of size servers (doubly linked)
  //this is done with a global array of channels

  AChannel<MType> *S_Chan[size], *C_Chan[size];

  parfor (int i=0; i<size; i++)
    S_Chan[i] = new AChannel<MType>;
  parfor (int j=0; j<size; j++) {
    C_Chan[j] = new AChannel<MType>;
    spawn Server (j, S_Chan[j], S_Chan[(j+size-1)%size],
                  S_Chan[(j+1)%size], C_Chan[j]);
    spawn Client (j, C_Chan[j], S_Chan[j]);
  }
}

void create_ring_it (void)
{
  //form a ring of size servers (doubly linked)
  //this is done with iteration

  AChannel<MType> *mine, *prev, *next, *client;
  AChannel<MType> *first_inC = new AChannel<MType>;
  AChannel<MType> *last_inC = new AChannel<MType>;
  mine = first_inC;
  prev = last_inC;
  for (int i=0; i<size; i++) {
    if (i<size-2) next = new AChannel<MType>;
```

```
      else if (i==size-2) next = last_inC;
      else                       /*(i==size-1)*/ next = first_inC;
    client = new AChannel<MType>;
    spawn Server (i, mine, prev, next, client);
    spawn Client (i, client, mine);
    prev = mine;
    mine = next;
  }
}

void create_ring_rec (int low, int high,
        AChannel<MType> *&LeftIn, AChannel<MType> *&RightIn,
        AChannel<MType> *LeftOut, AChannel<MType> *RightOut) {
  //form a ring of size servers (doubly linked)
  //this is done with recursion

  if (low > high) {
    LeftIn = RightOut;
    RightIn = LeftOut;
  }
  else {
    int mid = (low+high)/2;
    AChannel<MType> *myChan = new AChannel<MType>;
    AChannel<MType> *myLeft, *myRight;
    create_ring_rec (low, mid-1, LeftIn, myLeft,
                     LeftOut, myChan);
    create_ring_rec (mid+1, high, myRight, RightIn,
                     myChan, RightOut);
    AChannel<MType> * client = new AChannel<MType>;
    spawn Server (mid, myChan, myLeft, myRight, client);
    spawn Client (mid, client, myChan);
  }
}


void
main()
{
  printf ("Starting global array version....\n");
  create_ring_global();
  printf ("done.\n\n");

  printf ("Starting iterative version....\n");
  create_ring_it();
  printf ("done.\n\n");

  printf ("Starting recursive version....\n");
```

```
  AChannel<MType> *Loop = new AChannel<MType>;
  AChannel<MType> *Left, *Right;
  create_ring_rec(1,size-1,Left,Right,Loop,Loop);
  AChannel<MType> * client = new AChannel<MType>;
  spawn Server (0, Loop, Right, Left, client);
  spawn Client (0, client, Loop);
  printf ("done.\n\n");
}
```

**Discussion** In this example, each server has a unique channel on which it alone receives messages. Three other processes send messages on this channel: the next server in the ring sends requests for the token, the previous server in the ring sends the token, and the server's client sends requests to enter the critical section. This functionality is achieved by giving each writer a pointer to the server's channel. Having multiple writers to a single channel can be seen as a merge of multiple channels.

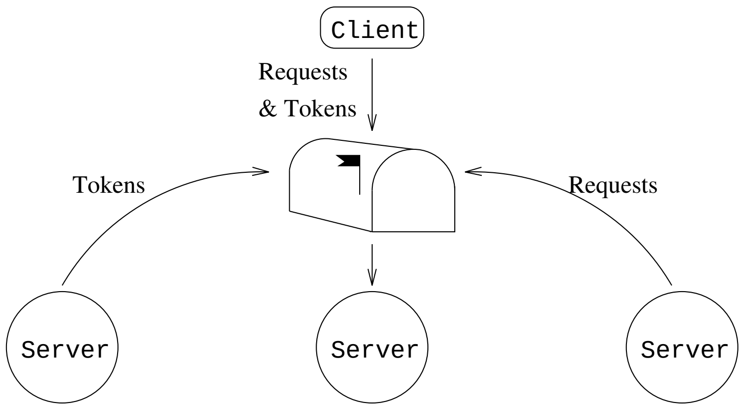


Figure 4.6: Multiple Writers to a Single Channel

Three functions are given which create the required structure. The first uses an array of channels. Server $i$ sends to channels $i-1$ and $i+1$. The second function creates the ring structure dynamically, using iteration. Notice how a channel is created once, and then pointers to the channel are manipulated. The third function uses recursion to dynamically create the ring structure. Each level of recursion creates a doubly linked list of channels and returns a pointer to the leftmost and rightmost of these channels. Again notice that no copying or explicit merging of channels is required.

## 4.7 Correctness

We show that the conjunction of equations (4.3) through (4.10) is an invariant of the program. The equations hold initially because `EmptySlots` and `Undelivered` are empty, and `cS`, `cR`, and `iR` are zero. Next, we show that the equations are maintained by providing the annotated programs for the member functions of the channel.

### Annotated Program for Give_Slot

```
template <class Message>
atomic void AChannel <Message>::give_slot(Message * sync * p)
{
  // BEGIN ASSERTIONS
  // (4.3) - (4.10)
  // END ASSERTIONS

  // iR++;

  // BEGIN ASSERTIONS
  // #EmptySlots = iR - cR - 1
  // (4.4) - (4.10)
  // END ASSERTIONS

  if (Undelivered.isempty()) {
    // BEGIN ASSERTIONS
    // #Undelivered = 0
    // #EmptySlots = iR - cR - 1
    // (4.4) - (4.10)
    // END ASSERTIONS

    EmptySlots.append(p);

    // BEGIN ASSERTIONS
    // #Undelivered = 0
```

```
    // (4.3) - (4.10)
    // END ASSERTIONS

  }
  else {
    // BEGIN ASSERTIONS
    // #Undelivered > 0
    // cR < cS
    // #EmptySlots = 0
    // #EmptySlots = iR - cR - 1
    // (4.4) - (4.10)
    // END ASSERTIONS

    p=Undelivered.dequeue();
    // cR++;

    // BEGIN ASSERTIONS
    // #EmptySlots = 0
    // (4.3) - (4.10)
    // END ASSERTIONS

  }
  // BEGIN ASSERTIONS
  // (4.3) - (4.10)
  // END ASSERTIONS

}
```

### Annotated Program for Nonblocking Send

```
template <class Message>
atomic void AChannel <Message>::nonblockingSend(Message &M)
{
  // BEGIN ASSERTIONS
  // s(cS) = M
  // (4.3) - (4.10)
  // END ASSERTIONS

  Message* q = new Message (M);
```

```
// BEGIN ASSERTIONS
// s_(cS) = *q
// (4.3) - (4.10)
// END ASSERTIONS

if (EmptySlots.isempty()) {
  // BEGIN ASSERTIONS
  // #EmptySlots = 0
  // s_(cS) = *q
  // (4.3) - (4.10)
  // END ASSERTIONS

  Undelivered.append(q);

  // BEGIN ASSERTIONS
  // #EmptySlots = 0
  // Undelivered[#Undelivered-1] = s_(cS)
  // #Undelivered = cS - cR + 1
  // (4.4), (4.6) - (4.10)
  // END ASSERTIONS

}
else {
  // BEGIN ASSERTIONS
  // #EmptySlots > 0
  // #Undelivered = 0
  // s_(cS) = *q
  // (4.3) - (4.10)
  // END ASSERTIONS

  EmptySlots.dequeue()=q;
  // cR++;

  // BEGIN ASSERTIONS
  // #Undelivered = 0
  // s_(cS) = slot_(cR-1)
  // cR ≤ cS + 1
```

```
    // #Undelivered = cS - cR + 1
    // (4.3), (4.4), (4.6), (4.8) - (4.10)
    // END ASSERTIONS

  }
  // cS++;

  // BEGIN ASSERTIONS
  // (4.3) - (4.10)
  // END ASSERTIONS

}
```

**Annotated Program for Blocking Receive**

```
template <class Message>
Message* AChannel <Message>::blockingReceive(void)
{
  // BEGIN ASSERTIONS
  // (4.3) - (4.10)
  // END ASSERTIONS

  Message * sync * p = new Message * sync;

  // BEGIN ASSERTIONS
  // (4.3) - (4.10)
  // END ASSERTIONS

  give_slot(p);

  // BEGIN ASSERTIONS
  // (4.3) - (4.10)
  // END ASSERTIONS

  return p;

  // BEGIN ASSERTIONS
  // (4.3) - (4.10)
  // END ASSERTIONS
```

}

## 4.8 Proof of Specification

**Proof of (4.1)** :

$$
\begin{array}{ll}
& \texttt{TRUE} \\
\Leftrightarrow & \quad \{ \text{ by 4.8, 4.3, and 4.5 } \} \\
& (\texttt{cR} = \texttt{iR}) \vee (\texttt{cR} = \texttt{cS}) \\
\Leftrightarrow & \quad \{ \text{ by 4.7 and definition of } \texttt{cR} \ \} \\
& \texttt{cR} \geq \min(\texttt{iR}, \texttt{cS}) \wedge (\texttt{cR} \leq \texttt{iR}) \wedge (\texttt{cR} \leq \texttt{cS}) \\
\Leftrightarrow & \quad \{ \text{ property of min } \} \\
& \texttt{cR} = \min(\texttt{iR}, \texttt{cS}) \quad \Box
\end{array}
$$

**Proof of (4.2)** : For any $k$ $(0 \leq k < \texttt{cR})$

$$
\begin{array}{ll}
& \texttt{r}_{(k)} \\
= & \quad \{ \text{ by 4.9 } \} \\
& \texttt{slot}_{(k)} \\
= & \quad \{ \text{ by 4.10 } \} \\
& \texttt{s}_{(k)} \quad \Box
\end{array}
$$

# Chapter 5

# A Transport Layer for Ported Asynchronous Channels

## 5.1 Introduction

A channel can be seen as a pair of ports, rather than as a single mailbox (as described in chapter 4). Sends are performed on the channel's out port and receives are performed on the channel's in port. The specification of such a ported channel is precisely the same as that of the `Channel` object described in chapter 4.

A transport layer is a collection of member functions and data members which are added to a ported channel. When a channel connects two processes in distinct memory address spaces, the transfer of the message data (in this case performed via an RPC call) is typically the most time consuming step in a `nonblockingSend` operation. The transport layer makes this cost virtually transparent to the calling process by performing the required message transfer outside of the `nonblockingSend` operation, and thus allowing this operation to terminate once a local copy of the message has been made. The capacity of this local buffer (as with the remote buffer) is determined by the memory limitations of the implementation.

This layer does not add any public member functions, nor any public data members, to the ported channel. The specification for this class, however, is modified to reflect the dissociation between the sending and the transmission of a message.

## 5.2 Specification

Let `cS` be the number of `nonblockingSend()` operations executed on the channel. Let `iR` and `cR` be the number of `blockingReceive()` operations that have initiated and have completed respectively.

The progress condition is that if at least $k$ `nonblockingSend()` operations have completed and at least $k$ `blockingReceive()` operations have been initiated, then at least $k$ `blockingReceive()` operations will terminate.

$$\forall k :: (\texttt{cS} \geq k) \wedge (\texttt{iR} \geq k) \leadsto (\texttt{cR} \geq k) \tag{5.1}$$

The first safety condition is that the number of `blockingReceive()` operations that have completed is bound by number of `blockingReceive()` operations that were initiated and the number of `nonblockingSend()` operations that have completed.

$$\texttt{cR} \leq \min(\texttt{iR}, \texttt{cS}) \tag{5.2}$$

In addition, the channel delivers the messages in order. Let $\texttt{s}_{(k)}$ be the message sent by the $(k+1)^{th}$ `nonblockingSend` executed on the channel, for $0 \leq k < \texttt{cS}$. Let $\texttt{r}_{(k)}$ be the message received by the $(k+1)^{th}$ `blockingReceive()` executed on the channel, for $0 \leq k < \texttt{cR}$. A second safety condition is that the $(k+1)^{th}$ message received is the $(k+1)^{th}$ message sent.

$$\forall k : 0 \leq k < \texttt{cR} : \texttt{r}_{(k)} = \texttt{s}_{(k)} \tag{5.3}$$

Note that this is the specification for a channel, which is a *pair* of ports.

## 5.3 Design

### 5.3.1 Channel as a Pair of Ports

A channel is divided into two objects: an `Out_Channel` on which `nonblockingSend()` operations are performed, and an `In_Channel` on which

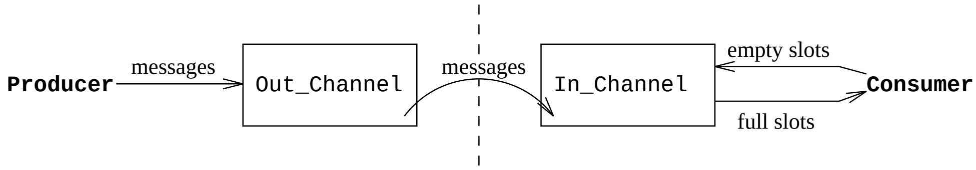


Figure 5.1: Division of Channel into an `Out_Channel` and an `In_Channel`

`blockingReceive()` operations are performed. Each `Out_Channel` corresponds to precisely one `In_Channel`. An `Out_Channel` contains a data member called `Dest` which is a pointer to the corresponding `In_Channel`.

The `Out_Channel` constructor must be passed the address of the `In_Channel` to which it is to be linked. Thus, an `Out_Channel` is associated with exactly one, and always the same, `In_Channel`.

An `In_Channel` object is exactly the `Channel` object described in chapter 4. We rename `Channel::nonblockingSend()` to `In_Channel:: hear()` to avoid confusion with the `nonblockingSend()` member of `Out_Channel`.

### 5.3.2 Message Transfer by RPC

The transfer of the message data of an `Out_Channel` is performed by a remote procedure call of the corresponding `In_Channel`'s `hear()` member function. The CC++ keyword `global` is used to indicate that the `Dest` pointer is valid across distinct memory address spaces. Also, the CC++ operators `<< void` and `>> void` are invoked for the message since it is a parameter in the RPC. For a more complete discussion of these CC++ constructs, see [15] and [12].

### 5.3.3 Local Copy of Messages

The purpose of the transport layer is to make the cost of the physical transfer of data to a remote (non-local) address space as transparent to the sending process as possible. Since an asynchronous send operation will not wait for an acknowledgement from the remote destination, its specification will often include the condition that on termination, the message has left the local node (and hence it is safe to deallocate this space). Thus, the transport layer must make a *local* copy of the message being sent before permitting the send operation to terminate. The sending process may then safely deallocate the original message, for it is the copy made by the transport layer that will (eventually) be transmitted. These local copies waiting to be transmitted are stored in a queue `Untransported`. Because of this local copying, a copy construct must be defined for the `Message` type if it requires any form of deep copying.

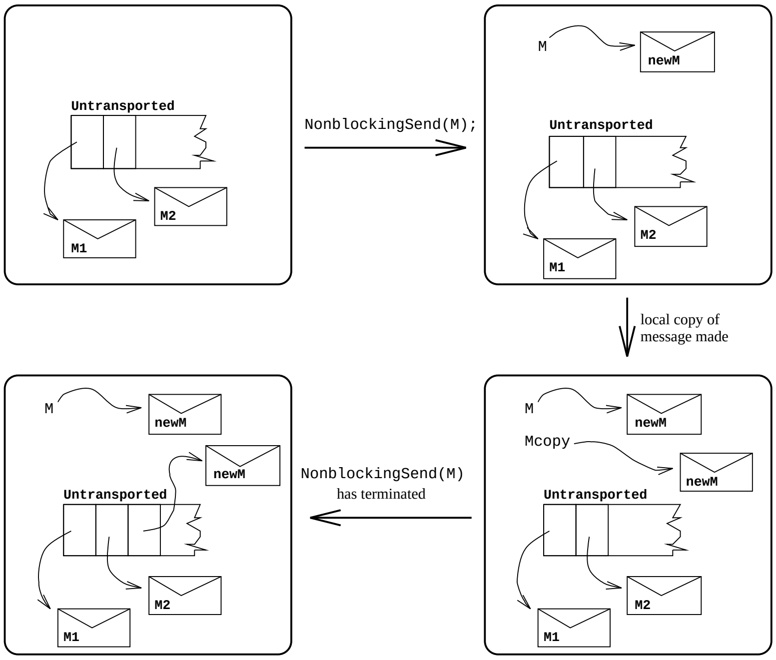


Figure 5.2: Local Copy of Message

### 5.3.4 Performance Considerations

Because the transport layer actually involves making an extra copy of each message, this optimization is appropriate only under certain circumstances. A large and complicated structure might have such a high cost associated with making the local copy that the transport layer would actually degrade performance. The situation is represented pictorially in Figure 5.3.

It is easy to see that if the objective function is to minimize the delay of the sending process, then the transport layer represents an improvement over the regular library whenever the time required for a local copy (labelled

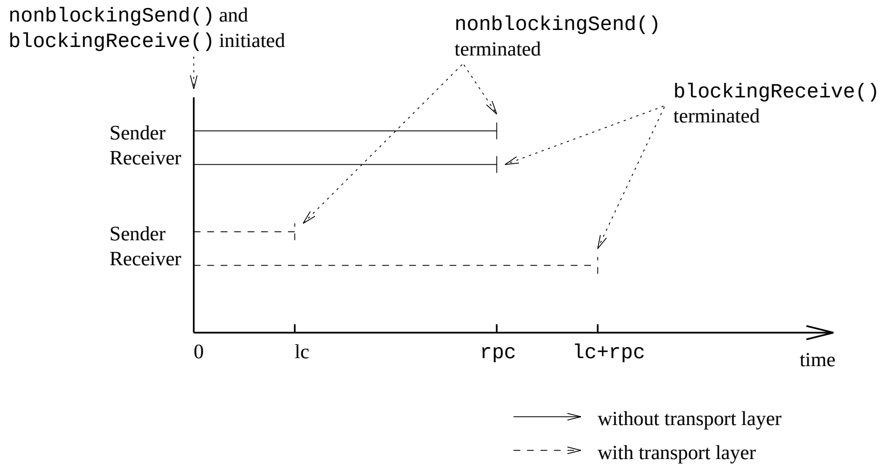


Figure 5.3: Time Cost Associated with Transport Layer

`lc` in Figure 5.3) is less than the time required for the RPC (labelled `rpc` in figure). On the other hand, if the objective function is to minimize the sum of the delay of the sending and receiving processes, then the transport layer might be justified only when `lc` < `rpc`/2. The programmer should be aware that many subtle issues, such as the distribution of `nonblockingReceive()` operations, synchronizations between the sending process and the receiving process, or the physical distribution of the computation, can greatly affect the performance of the transport layer and even determine whether it represents an improvement or a degradation.

### 5.3.5 Access to `Untransported` Queue

The principal problem to be solved is that of synchronization between the sending process (or *producer*) which is adding messages to the `Untransported` queue, and the transporting process (or *consumer*) which is removing messages from this queue. When the queue is empty, the transporting process should suspend, until a message is added by a sending process. Conversely, the transporting process should remain active so long as the queue is non-empty. (See Figure 5.4)

This is precisely the behaviour provided by a semaphore mechanism. The sending process issues a `V()` operation on the semaphore after having enqueued its message, and the transporting process issues a `P()` prior to dequeuing the message it will copy to the remote destination. The value of

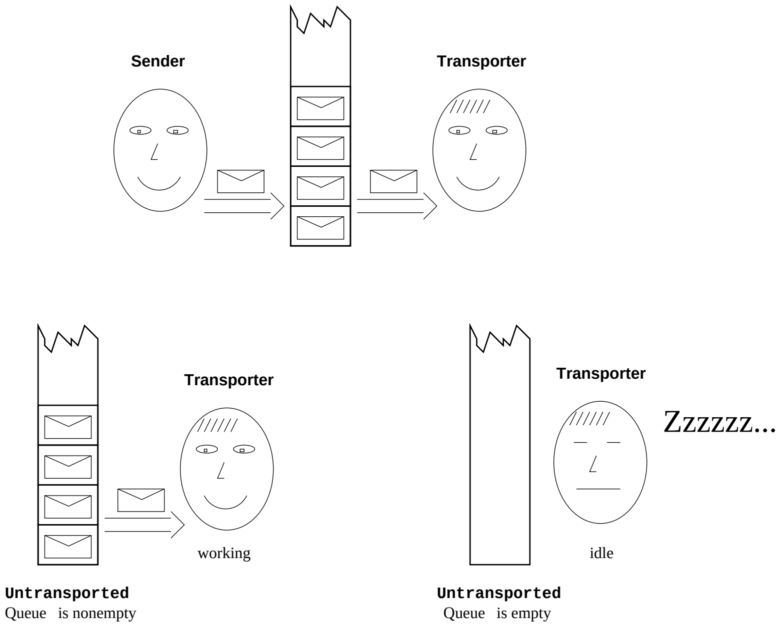


Figure 5.4: Interaction Between Sending and Transporting Processes via the `Untransported` Queue

this semaphore, then, is the number of elements contained in the queue.

## 5.4 Implementation State and Properties

In this section we give the invariants for the transport layer, which is entirely contained in the `Out_Channel` object. We will use these invariants in conjunction with the specification for the `In_Channel` object to prove the implementation meets the specification outlined in section 5.2.

The implementation of the `In_Channel` object is precisely the same as that given in chapter 4, but with the `nonblockingSend()` member function being named `hear()` instead. We can therefore use the proof in chapter 4 to conclude that the implementation of the `In_Channel` object meets the specification given in chapter 4. Replacing occurences of `cS` in this specification with `cH` (the number of completed `hear()` operations) and occurences of $\mathtt{s}_{(i)}$ with $\mathtt{h}_{(i)}$ (the $(i+1)^{th}$ message heard), we have the following specification for In_Channel:

$$\texttt{cR} = \min(\texttt{iR}, \texttt{cH}) \tag{5.4}$$

$$\forall k : 0 \leq k < \texttt{cR} : \texttt{r}_{(k)} = \texttt{h}_{(k)} \tag{5.5}$$

We make use of the semaphore object, as specified in chapter 2. Thus, we have:

$$\texttt{cP} \quad = \quad \min(\texttt{iP}, \texttt{s}_0 + \texttt{cV}) \tag{5.6}$$

$$\forall i : i \geq 1 : \texttt{terminated}(\texttt{P}_i) \quad \Rightarrow \quad \texttt{terminated}(\texttt{P}_{i-1}) \tag{5.7}$$

### 5.4.1 State

The state of the transport layer is given by:

1. A queue of untransported messages called Untransported. Recall cH is the number of RPC invocations of the In_Channel's hear() member function that have completed.

   $$\texttt{cS} \quad = \quad \#\texttt{Untransported} + \texttt{cH} \tag{5.8}$$

   where #Untransported is the number of elements in the Untransported queue.

   Messages are placed into, and retrieved from, the queue Untransported in order.

   $$\forall j : 0 \leq j < \#\texttt{Untransported} : \texttt{Untransported}[j] = \texttt{s}_{(\texttt{cH}+j)} \tag{5.9}$$

   where Untransported$[j]$ is the $(j+1)^{th}$ element of the Untransported queue.

2. The state of the semaphore used to synchronize the sending and the transporting process.

   $$\texttt{cS} \quad = \quad \texttt{cV} \tag{5.10}$$

### 5.4.2 Properties

**Ordering Requirement** The transport layer transmits the messages in order.

$$\forall k : 0 \leq k < \texttt{cH} : \texttt{s}_{(k)} = \texttt{h}_{(k)} \tag{5.11}$$

## 5.5 Implementation

```
template <class Message>
class Out_Channel {
  private:
    Queue<Message> Untransported;
    In_Channel<Message> * global Dest;
    Semaphore S;

    atomic void transmit(void);
    void transport(void);

  public:
    Out_Channel(In_Channel<Message>*global);
    atomic void nonblockingSend(Message);
};
```

**Constructor**

```
template <class Message>
Out_Channel<Message>::Out_Channel (In_Channel<Message> *
global d) :  S(0) {
  Dest = d;
  spawn transport();
}
```

**Transmit**

```
template <class Message>
atomic void Out_Channel<Message>::transmit(void) {
  Message * next_message = Untransported.dequeue();
  Dest→hear(*next_message);
  delete next_message;
}
```

**Transport**

```
template <class Message>
```

```
void Out_Channel<Message>::transport(void) {
  while (TRUE) {
    S.P();
    transmit();
  }
}
```

**Nonblocking Send**

```
template <class Message>
atomic void Out_Channel<Message>::nonblockingSend(Message &M)
{
  Message * q = new Message (M);
  Untransported.append(q);
  S.V();
}
```

## 5.6 Examples

### 5.6.1 Producer/Consumer

**Motivation** A pair of ports is linked to form a channel.

**Problem Description** As in chapter 4, two processes, a producer and a consumer, communicate over a channel. The producer creates data which the consumer then uses.

```
#define N 20                                //number of iterations

In_Channel<float> inC;
Out_Channel<float> outC(&inC);

void Producer (void) {
  float out_data;
  for (int i=0; i<N; i++) {
    out_data = produce_data();
    outC.nonblockingSend(out_data);
  }
}

void Consumer (void) {
```

```
  float * in_data;
  for (int i=0; i<N; i++) {
    in_data = inC.blockingReceive();
    consume_data(*in_data);
    delete in_data;
  }
}

main ()
{
  par {
    Producer();
    Consumer();
  }
}
```

**Discussion** The linking of ports is done at creation time of the `Out_Channel` object. Neither process need have any knowledge of the transport layer optimization being performed. The above code would not change if this optimization were not present.

Similarly, all the examples provided in chapter 4 could be repeated here. The only difference would be that each channel would consist of an `In_Channel` and an `Out_Channel` pair, linked together. It is important to note that this division of the `Channel` object corresponds to a modification in our view of a channel as a single mailbox, to a channel as a pair of ports. This modification does not represent a change in the public interface of a channel induced by the transport layer. The same code could be run on a channel library with or without this optimization.

## 5.7 Correctness

We show that the conjunction of equations (5.8) through (5.11) is an invariant of the program. The equations hold initially because `Untransported` is empty, and `cH` and `cS` are zero. Next, we show that the equations are maintained by providing the annotated programs for the relevant member functions of the `Out_Channel` class.

### Annotated Program for Transmit

```
template <class Message>
```

```
atomic void Out_Channel <Message>::transmit(void)
{
  // BEGIN ASSERTIONS
  // #Untransported > 0
  // (5.8) - (5.11)
  // END ASSERTIONS

  Message * next_message = Untransported.dequeue();

  // BEGIN ASSERTIONS
  // cS = #Untransported + cH + 1
  // $\forall j : 0 \leq j < \#\texttt{Untransported} : \texttt{Untransported}[j] = s_{\mathrm{cH}+j+1}$
  // next_message = $s_{(\mathrm{cH})}$
  // (5.10), (5.11)
  // END ASSERTIONS

  Dest->hear(*next_message);
  // cH++;

  // BEGIN ASSERTIONS
  // $s_{(\mathrm{cH}-1)} = h_{(\mathrm{cH}-1)}$
  // (5.8) - (5.11)
  // END ASSERTIONS

  delete next_message;

  // BEGIN ASSERTIONS
  // $s_{(\mathrm{cH}-1)} = h_{(\mathrm{cH}-1)}$
  // (5.8) - (5.11)
  // END ASSERTIONS
}
```

**Annotated Program for Transport**

```
template <class Message>
void Out_Channel <Message>::transport(void)
{
  // BEGIN ASSERTIONS
  // cH = 0
```

```
  // (5.8) - (5.11)
  // END ASSERTIONS

  while (TRUE) {

     // BEGIN ASSERTIONS
     // (5.8) - (5.11)
     // END ASSERTIONS

     S.P();

     // BEGIN ASSERTIONS
     // #Untransported > 0
     // (5.8) - (5.11)
     // END ASSERTIONS

     transmit();

     // BEGIN ASSERTIONS
     // #Untransported > 0
     // (5.8) - (5.11)
     // END ASSERTIONS

  }
}
```

**Annotated Program for nonblockingSend**

```
template <class Message>
atomic void Out_Channel <Message>::nonblockingSend(Message &M)
{
  Message * q = new Message (M);

  // BEGIN ASSERTIONS
  // s_(cS) = M
  // (5.8) - (5.11)
  // END ASSERTIONS

  Untransported.append(q)
```

```
  // BEGIN ASSERTIONS
  // Untransported[#Untransported-1] = M
  // s(cS) = M
  // #Untransported > 0
  // cS = #Untransported + cH - 1
  // (5.9) - (5.11)
  // END ASSERTIONS

  S.V();

  // BEGIN ASSERTIONS
  // Untransported[#Untransported-1] = M
  // s(cS) = M
  // #Untransported > 0
  // cS = #Untransported + cH - 1
  // cS = cV - 1
  // (5.9), (5.11)
  // END ASSERTIONS

  // cS++;

  // BEGIN ASSERTIONS
  // Untransported[#Untransported-1] = M
  // s(cS) = M
  // #Untransported > 0
  // (5.8) - (5.11)
  // END ASSERTIONS
}
```

## 5.8 Proof of Specification

**Lemma A** :

$$((\mathtt{cV} \geq k) \wedge (\mathtt{cP} = j < k)) \leadsto ((\mathtt{cV} \geq k) \wedge (\mathtt{cP} = j + 1))$$

Proof: We establish this result by showing that both of the following propositions are true:

1.

$$((\texttt{cV} \geq k) \wedge (\texttt{cP} = j < k)) \text{ unless } ((\texttt{cV} \geq k) \wedge (\texttt{cP} = j + 1))$$

This is true because stable($\texttt{cV} \geq k$) and because `cP` is monotonically increasing.

2.

$$\exists \text{transition } t :: \{(\texttt{cV} \geq k) \wedge (\texttt{cP} = j < k)\} t \{(\texttt{cV} \geq k) \wedge (\texttt{cP} = j + 1)\}$$

This transition is the `P()` operation in the `transport()` member function. This follows from the specification of the semaphore object ((5.6), (5.7)) and from the topology of the `transport() function`.

$\square$

**Lemma B** :

$$((\texttt{cV} \geq k) \wedge (\texttt{cP} < k)) \leadsto ((\texttt{cV} \geq k) \wedge (\texttt{cP} \geq k))$$

Proof: from Lemma A, using induction on $k - j$.

*Base case.* For $j = k - 1$

$$\begin{array}{ll} & (\texttt{cV} \geq k) \wedge (\texttt{cP} = j) \\ \leadsto & \quad \{ \text{ by Lemma A } \} \\ & (\texttt{cV} \geq k) \wedge (\texttt{cP} = k) \\ \Rightarrow & \quad \{ \ \} \\ & (\texttt{cV} \geq k) \wedge (\texttt{cP} \geq k) \end{array}$$

*Inductive hypothesis.* Suppose that for $0 \leq j < k - 1$

$$((\texttt{cV} \geq k) \wedge (\texttt{cP} = j + 1)) \leadsto ((\texttt{cV} \geq k) \wedge (\texttt{cP} \geq k))$$

*Inductive step.*

$$\begin{array}{ll} & (\texttt{cV} \geq k) \wedge (\texttt{cP} = j) \\ \leadsto & \quad \{ \text{ by Lemma A } \} \\ & (\texttt{cV} \geq k) \wedge (\texttt{cP} = j + 1) \\ \Leftrightarrow & \quad \{ \text{ by inductive hypothesis } \} \\ & (\texttt{cV} \geq k) \wedge (\texttt{cP} \geq k) \end{array}$$

$\square$

**Proof of (5.1)** :

$$
\begin{array}{ll}
 & (\mathtt{cS} \geq k) \wedge (\mathtt{iR} \geq k) \\
\Leftrightarrow & \quad \{ \text{ by } 5.10 \ \} \\
 & (\mathtt{cV} \geq k) \wedge (\mathtt{iR} \geq k) \\
\leadsto & \quad \{ \text{ by Lemma B } \} \\
 & (\mathtt{cP} \geq k) \wedge (\mathtt{iR} \geq k) \\
\leadsto & \quad \{ \text{ by topology of } \mathtt{transport()} \ \} \\
 & (\mathtt{cH} \geq k) \wedge (\mathtt{iR} \geq k) \\
\Leftrightarrow & \quad \{ \text{ property of min } \} \\
 & \min(\mathtt{cH}, \mathtt{iR}) \geq k \\
\Leftrightarrow & \quad \{ \text{ by } 5.4 \ \} \\
 & \mathtt{cR} \geq k \quad \Box
\end{array}
$$

**Proof of (5.2)** :

$$
\begin{array}{ll}
 & \mathtt{TRUE} \\
\Leftrightarrow & \quad \{ \text{ by } 5.4 \ \} \\
 & \mathtt{cR} = \min(\mathtt{iR}, \mathtt{cH}) \\
\Rightarrow & \quad \{ \text{ by } 5.8 \ \} \\
 & \mathtt{cR} = \min(\mathtt{iR}, \mathtt{cS}) \quad \Box
\end{array}
$$

**Proof of (5.3)** : For any $k$ such that $0 \leq k < \mathtt{cR}$,

$$
\begin{array}{ll}
 & \mathtt{r}_{(k)} \\
= & \quad \{ \text{ by } 5.5 \ \} \\
 & \mathtt{h}_{(k)} \\
= & \quad \{ \text{ by } 5.11 \ \} \\
 & \mathtt{s}_{(k)} \quad \Box
\end{array}
$$

# Chapter 6

# Conclusion

We have presented here a library integration of three imperative synchronization and communication paradigms in CC++: semaphores, monitors, and asynchronous channels. Each construct has been specified formally and implemented. The implementation of each library has been rigorously proven to meet its specification. Example programs have been given to illustrate the usefulness of these libraries and demonstrate the ease with which all of these concepts can be used in CC++.

It is interesting to note the brevity and simplicity of the proofs of correctness for these libraries. In each case the code was annotated with a few assertions to establish the invariance of a collection of properties, and then these properties were shown to imply the specification. Only predicate logic was required in each case, since the specifications were given entirely in terms of safety properties. This was possible because of the concept of atomicity in CC++. On the other hand, it appears that the full power of atomicity was not required. The degree to which a more limited version of atomicity is sufficient for general nondeterministic and reactive programs is an interesting topic worth further investigation.

Several extensions present themselves:

- The development and verification of libraries to support additional imperative synchronization and communication mechanisms, for example: rendezvous, synchronous message passing, conditional critical regions, and path expressions.
- The addition of libraries to support other parallel programming paradigms, for example: functional programming, logic programming, and

data parallelism.

- The creation of techniques for restricting nondetermism, for example: prohibiting more than one process from having access to the same channel or port.

We are encouraged by the preliminary examples that these verified libraries will be of great value to programmers.

# Bibliography


[1] Selim G. Akl, *The Design and Analysis of Parallel Algorithms*, Prentice Hall, 1989.

[2] Gregory R. Andrews, *Concurrent Programming: Principles and Practice*, Benjamin-Cummings, 1991.

[3] Stephen F. Barker, *The Elements of Logic*, McGraw-Hill, 1989.

[4] Lubomir Bic and Alan C. Shaw, *The Logical Design of Operating Systems*, Prentice-Hall, 1988.

[5] K. Mani Chandy and Jayadev Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[6] K. Mani Chandy and Stephen Taylor, *An Introduction to Parallel Programming*, Jones and Bartlett Publishers, 1992.

[7] Edsgar W. Dijkstra, *Cooperating sequential processes*, Mathematics Dept., Technological University, Eindhoven, The Netherlands, 1965.

[8] Per Brinch Hansen, *Operating System Principles*, Prentice-Hall, 1973.

[9] Hoare, C.A.R., Monitors: an operating system structuring concept, *Comm. ACM 17*, 10 (October), 549-557, 1974. Corrigendum, *Comm. ACM 18*, 2 (February), 95, 1975.

[10] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[11] Howard, J.H., Proving Monitors, *Comm. ACM 19*, 5 (May), 273-279, 1976.

[12] Carl Kesselman and K. Mani Chandy, *CC++: A Declarative Concurrent Object Oriented Programming Language*, CS-TR-92-01, California Institute of Technology, 1992.

[13] Harry R. Lewis, Christos H. Papadimitriou, C.H., *Elements of the Theory of Computation*, Prentice-Hall, 1981.

[14] Martin, A.J., An Axiomatic Definition of Synchronization Primitives, *Acta Informatica 16*, p219-235, 1981.

[15] Adam Rifkin, *A CC++ Primer for C Programmers*, CS-TR-93-14, California Institute of Technology, 1993.

[16] *The Hypercube Project Programmer's Manual*, JPL D-3220, October 30, 1989.