Concurrent Programming

Wim H. Hesselink, 23rd January 2007

Contents

1 Introduction

What is concurrency and why are we interested in it? The subject matter of this course is the art or craft of programming cooperating sequential processes. This problem originated in the construction of multi-user systems, see [7, 8].

Even on a PC with a single CPU, usually many processes are concurrently active, for example, the window manager, a word processor, a clock, the mail system, a printer driver. In this case, the processes are very different and their interaction mainly consists of competition for system resources. Indeed, they all have to wait for the CPU to serve them.

In embedded systems like refrigerators, cars or airplanes, physical machinery is controlled by microprocessors and the collaboration of the machinery requires communication of the processors. These processors may be regarded as concurrent processes. Their communication differs from the case of processes on a single CPU, but the problems and the solutions are rather similar.

In applications where heavy computations must be performed under tight time bounds, e.g., weather forecast or real-time image processing, the computational task can be divided among a number of concurrently running processors. Communication is required at least to distribute the problem and to collect the results. Usually the partioning of the problem is not absolute, in which case the subcomputations also need communication for their own purposes.

At the far end of the spectrum we have networks that consist of cooperating computers, say for distributed databases, the internet, etc.

The field of concurrency is often divided according to the communication medium. On the one hand, there are *shared memory* systems in which processes communicate by access to *shared variables*. The alternative is that processes communicate by *message passing*. In the case of message passing, there is the distinction between *synchronous* and *asynchronous* communication. Synchronous communication means that a process that needs to communicate is blocked if its partner is not yet ready for the communication. One sometimes uses the term distributed systems for concurrent systems with message passing.

The differences between these three kinds of concurrent systems are not very fundamental: in a shared memory system, one can emulate messages by means of buffers. Indeed, we shall model message passing architectures in terms of shared memory. Conversely, in practice, shared memory systems are sometimes implemented by means of messages, but this is harder. Since we regard shared memory as more fundamental than message passing, we'll start our investigations with shared memory.

The field of concurrency has extensions to real-time systems, fault-tolerant systems, and hybrid systems, but in this course we cannot deal with these extensions, apart from a small excursion to fault-tolerance.

1.1 Processes or threads

In these notes, we concentrate on the common abstraction of all these systems: they consist of *co-operating sequential processes*, see Dijkstra's paper [8] from 1968. The terms in computing science have evolved, however. By now, the smallest unit of sequential computation is usually called a *thread of control* or *thread* and the term *process* is used for a set of co-operating threads with combined control over some resources (like shared memory). This shift of terminology is recent and not yet universally accepted in the concurrency literature. Since control of resources is a matter of the operating system that usually can be separated cleanly from the issue of concurrent programming, we here use the terms process and thread as interchangeable synonyms.

A *thread* or *process* is a sequential program in execution. We are usually interested in a system of concurrent threads, i.e., threads that need to execute together.

A thread can be in three different states: it can be *running*, or *ready*, or *suspended*. When there is only one processor, at most one thread can be running, but there can be many ready threads. Similarly, on a multiprocessor system the number of running threads is bounded by the number of processors. A running thread that has to wait for some event is suspended. The scheduler then chooses another ready thread to run. A suspended thread is made ready again when the event happens that it is waiting for. From time to time the scheduler stops a running thread, makes it ready, and chooses another ready thread to run. In concurrent programming, we do not control the scheduler. We therefore often regard ready threads as running.

1.2 Challenge and responsibility

The challenge of concurrent programming is to ensure that never some part of the computations has to wait unnecessarily for another part. This means that, when the order of doing things is irrelevant, we prefer to leave the order unspecified by saying that it can be done in parallel, even on a uniprocessor system where things can never be done truly in parallel. The reason is greater flexibility: we want to postpone design decisions until they are unavoidable.

The concurrent programmer has a great responsibility for provable correctness. Firstly, concurrent programming is more difficult than sequential programming because of the danger of unexpected interferences between different processes (incorrect algorithms have been published in scientific papers).

Secondly, such unexpected interferences are often rare and are therefore not found by testing. They are often due to so-called race conditions, which means that the fault only occurs when one component has a certain speed relative to another component. This can be critically affected by print or assert statements in the code or by changes in the hardware. Therefore race conditions often remain hidden during testing.

Thirdly, in concurrency, the correctness issue is more complicated than for sequential programs. It starts with more complicated specifications that distinguish *safety* and *liveness* (or *progress*) properties. Safety properties express that nothing bad happens; liveness properties express that eventually something good happens. Safety properties are almost always expressed by invariants. If liveness properties are proved formally, this usually requires some form of variant functions.

Summarizing, correct concurrent programs are seldom obviously correct. So you always need to argue the correctness with some degree of formality. Indeed, formal correctness proofs can add significantly to the trustworthiness of a solution. One also uses model checkers and even theorem provers for these purposes.

Finally, the correctness of a concurrent program often depends on the assumptions about the concurrency involved. This is shown by the following example.

Example. Once we had a program with N concurrent processes, numbered from 0 to $N-1$ and an array $\texttt{b}[0\ .\ .\ N-1]$ of booleans. In some repetition, each process p had to set its own array element $\texttt{b}[p]$ to false after a part of its computation and then one process had to set all these booleans to true again. The program seemed to work correctly until the repetition was made longer to enable timing measurements. Then, once in a while, the system would go in deadlock.

After days of fruitless debugging, the reason was found. The booleans were implemented as bytes in the language C. The compiler assigned four bytes to a word, so the array was stored in $N/4$ words. Whenever a process modified its boolean, it did so by fetching a word from the array, modifying the appropriate byte and placing the word back again. We were therefore in trouble when two processes were modifying their booleans in the same word concurrently. □

1.3 The language SR

To illustrate the problems and the solutions, we use the programming language SR of [2], which is available on the WING system (your search path should contain `/opt/sr/bin`). This language is based on Dijkstra's guarded command notation. In particular, the repetition (**while** B **do** S **end**) is denoted in SR by

```
        do B -> S od .
```

The conditional statement in SR has the form

```
if B0 -> S0
 . . .
[] Bk -> Sk
[] else -> SE
fi
```

Execution of the **if** statement means execution of one of the commands `Si` for which the guard `Bi` evaluates to true. If none of the guards `Bi` holds, command `SE` is executed. The last branch can be omitted and then means that the **else** command is *skip* (do nothing).

In the language SR, it is easy to construct and test concurrent programs. Note however that, since the language is executed on a single processor, performance data for SR are irrelevant for the performance of a similar C–program on a multiprocessor system like the Cray.

1.4 The mutual exclusion problem as an example

As announced above, we start with a shared-memory system. A typical problem is that we have two concurrent processes that from time to time have to execute some task that requires exclusive access, say to some device. Let us call the processes A and B. The program fragment where they need exclusive access is called the *critical section* `CS`, the remainder of their programs is called the *noncritical section* `NCS`. The processes are supposed to execute many pairs `NCS` followed by `CS`. So we model them as consisting of an infinite loop. Note however, that `NCS` need not terminate, for it may be that a process is no longer interested in the device. On the other hand, we assume that `CS` does terminate and we want to guarantee that whenever a process needs to execute `CS`, it gets a chance to do so eventually. Initially, the code for the two processes looks as follows:

```
process A                  process B
   do true ->                 do true ->
      NCS ; CS                   NCS ; CS
   od                         od
end A                      end B
```

This is indeed SR, apart from the lay-out and the names `NCS` and `CS`.

Now, the problem is to ensure mutual exclusion: the processes are never at the same time in their critical sections. This property is expressed formally in the requirement

(MX) $\neg(A \textbf{ in } \texttt{CS} \quad \wedge \quad B \textbf{ in } \texttt{CS})$.

This predicate must always hold. In other words, it must be an invariant. This is a safety property. It could be affected by never allowing B to execute `CS`. So, additionally, we require the liveness property that, if a process needs to execute `CS`, it will do so eventually (it is not blocked indefinitely). This property is called *eventual entry* (EE).

This problem is called the *mutual exclusion* problem (dutch: wederzijdse uitsluiting). Before developing solutions to this particular problem, we first have to investigate the tools that are available and the properties of these tools.

Remark. For the moment, we use the convention that an invariant must always hold. This is one of the aspects where concurrency differs from sequential programming (the invariant of a repetition in sequential programming has to hold only at every inspection of the guard of the repetition, but may be falsified temporarily in the body). Later on, we shall be more liberal, but then we must very explicitly express where the invariants are supposed to hold. □

1.5 The execution model and atomicity

In order to argue about a shared variable system, we have to define first the *state* of the system and then the possible evolutions of the system under actions of the processes.

The *state* of a shared-variable system consists of

- the values of the shared variables
- for each process, the values of the private variables, including the program counter *pc* which indicates the instruction that is to be executed next.

When arguing about a concurrent program, we use the convention that shared variables are always in typewriter font, that private variables are slanted, and that the value of the private variable x of process p is denoted by $x.p$. In particular, $pc.p$ is the program counter of process p. We write p **at** i for $pc.p = i$. Note that when p is **at** i, label i is the label of the instruction that thread p will execute when it is scheduled *next* (we never argue about an atomic action that is being executed *now*).

An *execution sequence* of the system is a sequence of pairs $(x_0, p_0), \ldots (x_n, p_n), \ldots$, where each x_i is a state of the system and each p_i is a process that can execute in state x_i an *atomic action* with final state x_{i+1}.

This definition is based on the concept of *atomic or indivisible actions.* An action is the execution of one or more instructions. An action is *atomic* if concurrent actions by other processes **can not** lead to interference. The question which actions can be treated as atomic depends on the hardware (or, as in the case of SR, the underlying software).

The definition allows execution sequences in which some of the processes never act. Such an execution sequence may be regarded as "unfair". More precisely, an infinite execution sequence is defined to be *fair* if every process acts infinitely often in it.

Example. Suppose we have two shared integer variables `x` and `y`, initially 5 and 3, and we want to increment them concurrently. In SR, this is expressed by

```
# initial state: x = 5 and y = 3
co  x := x + 1  //  y := y + 1  oc
# final state: x = 6 and y = 4
```

This specifies that both are incremented, but not which of the two is incremented first. This code is completely unproblematic.

Suppose now, however, that we want to increment `x` twice, concurrently, by means of

```
# x = 5
co  x := x + 1  //  x := x + 1  oc
# x = ?
```

One may expect the postcondition `x = 7`, but this is not always guaranteed. Indeed, the two incrementations may be executed as follows: first, the lefthand process reads `x` and writes 5 in a register. Then the righthand process reads `x` and writes 5 in its register. Then both processes increment their registers and write 6 to the shared variable `x`. □

This example illustrates the question of the *grain of atomicity*: what are the atomic (indivisible) actions? Unless otherwise specified, we assume that an action on shared variables can (only) be atomic if no more than one shared variable is read <u>or</u> written in it. So, the incrementation `x:= x+1` is not atomic. If other processes may update `x` concurrently, an incrementation of `x` must be split up by means of a private variable. It is allowed to combine an atomic action atomically with *a bounded number of actions on private variables that do not occur in the specification of the system.* We come back to this later.

If S is a command, we write $\langle\ S\ \rangle$ to indicate that command S must be executed without interference by other processes. The angled brackets $\langle\rangle$ are called atomicity brackets. If command S happens not to terminate, the action does not occur and the executing process is blocked. Other processes may proceed, however, and it is possible that they would enable the blocked process. Thus, possibly nonterminating atomic commands are points where a process waits for actions of other processes.

1.6 Simple await statements

The most important possibly-nonterminating command is the *simple await statement* **await** B for some boolean expression B. Command **await** B is defined to be the atomic command that tests B and only proceeds if B holds. More precisely, if process p executes **await** B in a state x where B is false, the resulting state equals x and, since even $pc.p$ is unchanged, process p again is about to execute its await statement. The command is therefore equivalent to

$$\mathbf{do}\ \neg B \quad \rightarrow \quad \mathit{skip}\ \ \mathbf{od}\ ,$$

at least if guard B is evaluated atomically. Indeed, the straightforward way to implement await statements is by busy waiting. On single processor systems, busy waiting is a heavy burden on the performance. Later on, we therefore present some alternatives for the await statement.

If all processes are waiting at an await statement with a false guard, the system is said to be in *deadlock*. The absence of deadlock is a desirable safety property, always implied by the relevant liveness properties of the system. Since it is usually much easier to prove, we are sometimes content with proving deadlock-freedom rather than liveness.

1.7 Atomicity and program locations

When analysing a concurrent program, we often have to be very careful about the location where each process is. Indeed, this determines the instruction it will execute next although it does not determine when this instruction is executed. We can then number the program locations, i.e., label them. If we do so, we regard every labelled instruction as atomic and omit the atomicity brackets.

If process q is at location i, the instruction labelled i is its next instruction. By executing instruction i the process goes by default to location $i+1$. If the instruction at location i is the test of an **if**, **do**, or **await** statement, the new location of the process is determined in the obvious way. For example, assume process q is given by the infinite loop

```
         do true ->
3:         do ba ->
4:           out := false
5:           if bb ->
6:             await bc
7:             x ++
             fi
8:           out := true
           od
9:         y --
         od
```

If process q is **at** 3 and if it takes a step, it tests the value of `ba`. If `ba` holds, it goes to 4; otherwise, it goes to 9. If q **at** 5 takes a step, it goes to 6 if `bb` holds and it goes to 8 if `bb` is false. If q **at** 6 takes a step while `bc` holds, it goes to 7. If it takes a step when `bc` is false, it remains at 6. If q **at** 8 takes a step, it decrements `y` and goes to 3.

If L is a set of locations, we write q **in** L to denote that process q is at some of the locations in L.

Exercise 1. In the above program, assume that *out* is a private variable of process q with the initial value *true*. Argue that

$$\mathit{out}.q \quad \equiv \quad q\ \mathbf{in}\ \{3,4,9\}$$

holds initially and remains valid under each step of process q, and is therefore an invariant of the program.

2 Busy waiting

In this section we use the await statement introduced in 1.5 for the mutual exclusion problem of section 1.4 and also for the so-called barrier problem. For implementation, recall that **await** B can be implemented by busy waiting if B can be tested atomically.

2.1 Peterson's mutual exclusion algorithm

Solutions of the mutual exclusion problem always insert so-called entry and exit protocols before and after the critical section. If one thread is in the critical section and the other one needs to enter, the other one has to wait. So, what is more natural than introducing for each thread a shared variable to indicate that the thread is in its critical section. We number the threads 0 and 1 and introduce a shared boolean array by means of the declaration and initialization:

```
         var active [0:1]: bool := ([2] false)
```

As a first attempt, we consider two threads declared by

```
(0)      process ThreadMX (self:= 0 to 1)
            do true ->
10:            NCS
11:            active[self] := true
12:            await not active[1 - self]
13:            CS
14:            active[self] := false
            od
         end ThreadMX
```

We have labelled the program locations for easy reference, starting from 10.

Remark. We have also numbered `NCS` and `CS`. This is allowed since `NCS` and `CS` do not interfere with all actions on the relevant variables and may therefore be treated as atomic. `NCS` need not terminate. This is modelled by assuming that, when a thread executes `NCS`, it either goes to 11 or goes back to 10. □

We now have to show that this program establishes mutual exclusion. This amounts to proving the following invariant:

(MX) $\neg(0 \textbf{ at } 13 \ \wedge \ 1 \textbf{ at } 13)$.

In order to do this, we first observe that, since `active`$[q]$ is only modified by thread q, it is easy to verify the invariant property, for both $q = 0$ and $q = 1$,

(I0) `active`$[q] \ \equiv \ q \textbf{ in } \{12, 13, 14\}$.

Indeed, predicate (I0) holds initially, since then `active`$[q]$ is false and q is **at** 10. Whenever thread q enters the region $\{12, 13, 14\}$, it makes `active`$[q]$ true, and vice versa. Whenever thread q exits the region $\{12, 13, 14\}$, it makes `active`$[q]$ false, and vice versa. This shows that (I0) is an invariant of the system.

We now show that (MX) is an invariant. It holds initially since then both threads are at 10. Whenever one thread, say p, establishes p **at** 13, it steps from 12 to 13 and thus verifies that, in that state, $\neg$`active`$[q]$ holds for $q = 1 - p$. By (I0), this implies that q is not **at** 12, 13, 14. In particular, q is not **at** 13. This proves that (MX) is an invariant.

Unfortunately, the above solution is incorrect since it can reach deadlock.

Exercise 1. Show that system (0) can reach deadlock, when it starts in a state where both threads execute `NCS`.

The remedy Peterson invented [14], was to introduce a shared variable `last` to indicate the thread that was the last one to enter its entry protocol.

```
(1)      process ThreadMX (self:= 0 to 1)
            do true ->
10:            NCS
11:            active[self] := true
12:            last := self
13:            await (last = 1-self or not active[1-self])
14:            CS
15:            active[self] := false
            od
         end ThreadMX
```

Note that action 13 inspects two different shared variables. For the moment, we assume that this is allowed. In the same way as before, we have the invariant

(J0) $\texttt{active}[p] \quad \equiv \quad p \textbf{ in } \{12, 13, 14, 15\}$.

Mutual exclusion is now expressed by the required invariant

(MX) $\neg(0 \textbf{ at } 14 \quad \wedge \quad 1 \textbf{ at } 14)$.

To show that this is indeed an invariant, we investigate how it can be violated by an action of thread p. In that case, the action must start in a state where $q = 1 - p$ is **at** 14 and p is **at** 13. Since p goes from 13 to 14 and $\texttt{active}[q]$ holds because of (J0), we have $\texttt{last} \neq p$. In order to exclude this possibility, it suffices to postulate the new invariant (for all p, q)

(J1) $p \textbf{ at } 13 \quad \wedge \quad q \textbf{ at } 14 \quad \Rightarrow \quad \texttt{last} = p$.

We now have to show that (J1) is invariant. Well, there are only two threads. If an action of one of them would violate (J1), it would be either p going to 13 or q going to 14. If thread p goes to 13, it executes action 12 and establishes $\texttt{last} = p$. If thread q goes to 14, it executes 13 with $\texttt{last} \neq q$ because of (J0), and hence with $\texttt{last} = p$.

It is easy to prove that system (1) is deadlock free. One just assumes that deadlock would occur and then analyses the state:

$$\begin{array}{ll}
 & \text{deadlock} \\
\equiv & \{ \text{ the only waiting is at 13 } \} \\
 & \text{0 and 1 are both blocked at 13} \\
\Rightarrow & \{ \text{ see the test in the await statement at 13 } \} \\
 & \texttt{last} = 0 \quad \wedge \quad \texttt{active}[1] \quad \wedge \quad \texttt{last} = 1 \quad \wedge \quad \texttt{active}[0] \\
\equiv & \{ \; 0 \neq 1 \; \} \\
 & \textit{false} \; .
\end{array}$$

There are no states where *false* holds. So deadlock does not occur.

Liveness is more than just the absence of deadlock. In system (1), liveness means that, if thread p is **at** 11, it will arrive **at** 14 within a bounded number of *rounds*. Here, we define a *round* to be a finite execution in which every thread acts at least once.

So, to prove liveness, we assume that p is **at** 11. After two rounds, p is **at** 13, if it has not yet reached 14. If p has not reached 14 after three rounds, then p has tested that $\texttt{last} = p$ and that $q = 1 - p$ satisfies $\texttt{active}[q]$. If p has not reached 14 after seven rounds, then q has proceeded beyond its critical section and executed action 15. If p has not reached 14 after eight rounds, q has reentered its entry protocol by executing 11. If p has not reached 14 after nine rounds, we have $\texttt{last} = q$. Therefore, p reaches location 14 within ten rounds.

Note that we did not assume (or use) that q executes `NCS` in one round. In fact, q may remain executing `NCS` forever, but then p reaches `CS` more easily.

Remark. It has been said that the correctness of Peterson's algorithm was sheer luck. To illustrate this, let us swap the actions 11 and 12 in (1) and see whether mutual exclusion still holds. We can try to give a proof or to construct a counterexample. We do the latter by giving a so-called *scenario*, i.e., an execution that violates the specification. The threads 0 and 1 enter almost concurrently. First 0 executes 11, so that `last` = 0. Then 1 executes 11, so that `last` = 1. Then 1 executes 12, so that `active`[1] holds. Then 1 passes 13, since `active`[0] is still false. Then 0 executes 12, so that `active`[0] holds. Then 0 passes 13 since `last` = 1. Now, both 0 and 1 are **at** 14, thus violating mutual exclusion. □

Exercise 2. Consider the variation of (1) where the second disjunct in the guard of 13 is removed. Show that mutual exclusion and deadlock-freedom still hold, but that the algorithm is nevertheless incorrect.

2.2 Mutual exclusion between more threads

Now suppose that we have $N > 2$ threads that repeatedly and concurrently execute noncritical sections and then try to get exclusive access to some resource. We then have the N–thread mutual exclusion problem. The book [1] gives a solution for this problem in which each thread in its entry protocol traverses $(N-1)^2$ await statements. This is unnecessarily inefficient.

The following alternative seems to be well-known, see [17]. The idea is to organize a tournament to decide which thread is allowed to enter its critical section. For simplicity, we assume $N = 2^K$. Organize the threads as the leaves of a balanced binary tree of depth K. Whenever a thread needs to enter `CS`, it starts at its leaf and executes K entry protocols of Peterson's algorithm to enter its critical section; it then executes K exit protocols to return at `NCS`.

Exercise 3. Programming: implement this idea in SR. A test version of Peterson's algorithm for two threads is made available via the Web. Generalize this program to $N = 2^K$ threads, numbered from 0 to $N-1$. Do not accumulate the trace, but let each thread write its number when it executes `CS`. □

2.3 Formal definitions for safety and progress

Now that we have seen some proofs of safety and progress, it is useful to formalize the concepts we are using.

For predicates X and Y, we say that X implies Y, notation $[X \Rightarrow Y]$, if at every state where X holds, Y holds as well. We write $X \rhd Y$ (pronounced: X step Y) to denote that every step of the program taken in a state where X holds terminates in a state where Y holds. Using this concept, we define a *strong invariant* to be a predicate X that holds initially and satisfies $X \rhd X$. For, clearly, such a predicate holds in all reachable states. Every predicate implied by a strong invariant is an invariant, but not every invariant is a strong invariant.

Example. Consider a system of a single process and one variable, and one nondeterministic infinite loop

```
        var $x : \{0,1,2,3,4\} := 0$ ;
(*)     do  $x = 0 \quad\rightarrow\quad x := 2$
        [] $x < 2 \quad\rightarrow\quad x := x + 3$
        [] $true \quad\rightarrow\quad skip$
        od .
```

The only location is (*). The state is therefore characterized by the value of x. The reachable values are 0, 2, 3. The predicate $x \neq 1$ is a strong invariant, since there is no step that can invalidate this. The predicate $x < 4$ is an invariant since it holds in all reachable states. It is not a strong invariant, however, since it holds at $x = 1$ where it can be invalidated by the second alternative. □

The liveness relation $o\!\!\rightarrow$ is defined as follows. Recall from section 2.1 that a *round* is a finite execution in which every thread acts at least once. In this definition, we take it that a thread at a nonterminating action (like `await` or `NCS`) can perform a *skip* action that does not modify the state in any way. For predicates X and Y, we define $X\ o\!\!\rightarrow\ Y$ (pronounced: X leads to Y) to mean that there is a number k such that every execution that is a concatenation of k rounds and that starts in a state satisfying X, also contains a state that satisfies Y.

Example. Liveness for Peterson's mutual exclusion algorithm was expressed by stating that, whenever a thread p is **at** 11, it will arrive **at** 14 within a bounded number of rounds. We can now express this by: p **at** 11 $o\!\!\rightarrow$ p **at** 14. □

2.4 Mutual exclusion formalized

We can now give a formal specification of a mutual exclusion protocol. The parameters of the problem are a number N and two program fragments `CS` and `NCS`, such that `CS` necessarily terminates, i.e., satisfies q **in** `CS` $o\!\!\rightarrow$ q **not-in** `CS`. There is no such condition for `NCS`. A mutual exclusion protocol for N threads with critical section `CS` and noncritical sections `NCS` consists of program fragments `Entry` and `Exit` such that the thread declaration

```
process ThreadMX (self := 0 to N-1)
  do true ->
     NCS ; Entry ; CS ; Exit
  od
```

satisfies the invariant (MX) and the liveness property (EE), which are given by

(MX) q **in** `CS` $\wedge$ r **in** `CS` $\Rightarrow$ $q = r$,
(EE) q **at** `Entry` $o\!\!\rightarrow$ q **at** `CS` .

Strictly speaking, there is also the requirement that `Exit` always terminates:

(ET) q **in** `Exit` $o\!\!\rightarrow$ q **not-in** `Exit` .

Since it is unlikely that protocols are proposed that violate it, condition (ET) is usually taken for granted.

Exercise 4. Which of the following predicates in section 2.1 is a strong invariant: (J0), (MX), (J1), (J0)$\wedge$(J1)? Give a strong invariant that implies all these predicates.

Exercise 5. In 1998, Yih-Kuen Tsay has refined Peterson's algorithm in the following way [16]. The program is:

```
         var
            active [0:1] := ([2] false)
            last := 1

         process Tsay (self := 0 to 1)
            do true ->
10:            NCS
11:            active [self] := true
12:            last := self
13:            if active [1-self] ->
14:               await last = 1-self
               fi
15:            CS
16:            active [self] := false
17:            last := self
            od end Tsay
```

(a) Show that the program satisfies mutual exclusion (MX) by first proving invariants analogous to (J0) and (J1) of section 2.1.
(b) For the sake of progress, prove an invariant of the form

(Pr) $\quad q$ **at** 14 $\;\Rightarrow\;$ $(1-q)$ **in** $\{\cdots\}$ $\;\vee\;$ `last` $= \cdots$.

Give a strong invariant that implies (Pr). Prove validity of q **at** 14 $o\!\!\rightarrow$ q **at** 15, and use this to prove eventual entry: q **at** 11 $o\!\!\rightarrow$ q **at** 15.

2.5 Barrier synchronization

When a task is distributed over several threads, it is often the case that, at certain points in the computation, the threads must wait for all other threads before they can proceed with the next part of the computation. This is called *barrier synchronization*, cf. [1]. We model the problem by letting each thread execute the infinite loop

```
do true ->
   TNS
   Barrier
od
```

Here, `TNS` stands for a terminating noncritical section, a program fragment that satisfies

(TN) $\quad q$ **in** `TNS` $\;o\!\!\rightarrow\;$ q **not-in** `TNS` ,

but that in all other aspects is irrelevant for the problem at hand.

The threads may only pass the barrier when all of them have completed the last noncritical section. On the other hand, when all of them have completed the last noncritical section, they all must pass the barrier and start the next noncritical section.

Thus, if we count the noncritical sections each thread has executed, the safety requirement is that the counters of the threads are always almost equal. In order to make this more precise, we provide each thread with a private ghost variable *cnt*, initially 0. The private variables *cnt* must not be modified by `TNS` and must be incremented by `Barrier` according to the Hoare triple

(H) $\quad \{\; cnt.self = C \;\}$ `Barrier` $\{\; cnt.self = C+1 \;\}$.

We now specify the barrier by the requirement that a thread that runs ahead with *cnt*, must not execute `TNS` but wait. This is expressed in the *barrier condition* that, for all threads q and r,

(BC) $\quad q$ **in** `TNS` $\;\Rightarrow\;$ $cnt.q \le cnt.r$.

Remark. One could propose to specify the barrier by the requirement that $cnt.q = cnt.r$ whenever both q and r are in `TNS`. This condition, however, would also be satisfied by disallowing different threads in `TNS` at the same time (i.e., by forcing mutual exclusion) and, then, no relationship between the counters of different threads would remain.

It is easy to see that (BC) implies that $cnt.q = cnt.r$ whenever both q and r are in `TNS`. Note that the ghost variables $cnt.q$ only serve in the formal specification, but that they have no role in the computation. □

We now turn to a solution. We assume that there are N threads, numbered 0 to $N-1$. It is clear that $N-1$ threads must wait at the barrier until the last one arrives. If we allow the ghost variables $cnt.q$ to enter the computation and to be inspected by the other threads, we could implement the barrier of *self* by means of a for-loop (`fa..af` in SR, where `st` means "such that")

```
cnt ++
fa i := 0 to N-1 st i != self  ->  await (cnt.i != cnt.self - 1)  af
```

When all threads arrive at the barrier at the same time, this may give memory contention for the threads at *cnt*.0. We therefore decide that the threads inspect the counters each in its own order, i.e., we give each thread a private variable `set` for a set of thread numbers (this is no SR) and let them choose nondeterministically:

```
cnt ++
var set := {0 .. N-1} \ {self}
do is_not_empty(set) ->
   extract some i from set ; await (cnt.i != cnt.self - 1)
od
```

Ghost variables can safely be unbounded integers, but in an actual program the integers are usually bounded somehow. Since the counters keep in step, however, it is allowed to compute them modulo some large number like `maxint`. Actually, we can also take the counters modulo some small constant $R \geq 3$. Let us represent them in an integer array `tag`. We want to avoid unnecessary inspection of shared variables. Each thread therefore gets a copy of its own previous `tag` value. In this way, we arrive at the following solution

```
(2)     var tag[0:N-1]: int := ([N] 0)

        process BarMember (self := 0 to N-1)
           var cnt := 0
           var set := empty_set
           do true ->
10:           TNS
              var old := tag [self]
11:           tag [self] := (old + 1) mod R ; cnt ++
              set := {0 .. N-1} \ {self}
12:           do is_not_empty(set) ->
                 choose i in set
13:              await (tag[i] != old)
                 set := set - {i}
              od
           od
        end BarMember
```

Since we have used a number of plausible transformations to arrive at this solution, we discuss its correctness separately. The elements of `tag` are the only shared variables. We have grouped and numbered the atomic actions in such a way that each of them contains only one inspection or update of a shared variable. For safety, we need to prove the invariant (BC). Clearly, (BC) is implied by the requirement

(K0) $\quad q \textbf{ in } \{10, 11\} \quad \Rightarrow \quad cnt.q \leq cnt.r\ .$

Since q arrives at 10, when it concludes the loop in $\{12, 13\}$, we need to generalize (K0) to a loop invariant that implies (K0) when *set.q* is empty. We thus generalize (K0) to the invariant

(K1) $\quad r \notin set.q \quad \Rightarrow \quad cnt.q \leq cnt.r\ .$

Predicate (K1) implies (K0) because of the obvious invariant

(K2) $\quad q \textbf{ in } \{10, 11\} \quad \Rightarrow \quad set.q = \emptyset\ .$

The properties (K1) and (K2) hold initially since we have chosen the initial values $cnt.p = 0$ and $set.p = \emptyset$. Property (K1) can only be violated by modification of *set.p* in actions 11 and 13, and of *cnt.p* in action 11. If thread p executes 11, it preserves (K1) since it puts all thread numbers different from p into *set.p*. If thread p executes 13, property (K1) is threatened only if $\texttt{tag}[i] \neq old$. It is preserved if we postulate the additional property

(Ka0) $\quad q$ **at** 13 $\;\wedge\;$ $\mathtt{tag}[r] \neq \mathit{old}.q \;\Rightarrow\; \mathit{cnt}.q \leq \mathit{cnt}.r$.

To prove (Ka0), we invent three invariants:

(K3) $\quad \mathtt{tag}[q] = \mathit{cnt}.q \;\mathbf{mod}\; R$,
(K4) $\quad q$ **in** $\{12, 13\} \;\Rightarrow\; \mathtt{tag}[q] = (\mathit{old}.q + 1) \;\mathbf{mod}\; R$,
(K5) $\quad \mathit{cnt}.q \leq \mathit{cnt}.r + 1$.

Since the values of $\mathtt{tag}[q]$, $\mathit{cnt}.q$, $\mathit{old}.q$ are only modified by thread q, it is easy to see that (K3) and (K4) are indeed invariant. Property (K5) can only be violated when thread q executes 11. Invariant (K0), however, implies that action 11 has the precondition $\mathit{cnt}.q \leq \mathit{cnt}.r$, so then it does not violate (K5).

It remains to show that (Ka0) is implied by (K3), (K4), (K5). This is shown in

q **at** 13 $\;\wedge\;$ $\mathtt{tag}[r] \neq \mathit{old}.q$
$\Rightarrow$ { (K4) and arithmetic; $\mathtt{tag}[r]$ and $\mathit{old}.q$ are in $[0\,.\,.\,R)$ }
$(\mathtt{tag}[r] + 1) \;\mathbf{mod}\; R \neq \mathtt{tag}[q]$
$\Rightarrow$ { (K3) for r and q, arithmetic }
$(\mathit{cnt}.r + 1) \;\mathbf{mod}\; R \neq \mathit{cnt}.q \;\mathbf{mod}\; R$
$\Rightarrow$ { arithmetic }
$\mathit{cnt}.q \neq \mathit{cnt}.r + 1$
$\Rightarrow$ { (K5) }
$\mathit{cnt}.q \leq \mathit{cnt}.r$.

Remark. We have used (K0) to prove invariance of (K5). This may give the impression of cyclic reasoning. The reasoning is sound, however, for we only use that all invariants hold in the precondition of a step, to prove that all of them also hold in the postcondition of the step. Here we use (K0) in the precondition of action 11 to prove that (K5) holds in the postcondition. □

Liveness of the barrier is expressed by the condition that the minimum of the values $\mathit{cnt}.x$ grows indefinitely. So, if we write $\mathit{MinC} = (\mathtt{MIN}\; x :: \mathit{cnt}.x)$, liveness is expressed in

(LB) $\quad \mathit{MinC} = X \quad o\!\!\longrightarrow \quad \mathit{MinC} > X$.

Relation (LB) is proved as follows. Assume $\mathit{MinC} = X$. Then, there is a thread q with $\mathit{cnt}.q = X$. Since the values of $\mathit{cnt}.x$ can only grow, and there are only N threads, it suffices to show that

(LBq) $\quad \mathit{MinC} = X = \mathit{cnt}.q \quad o\!\!\longrightarrow \quad \mathit{cnt}.q > X$.

If q is at 10 or 11, this $o\!\!\longrightarrow$ relation is obvious, since q will execute 11 within one or two rounds. So, we may assume that q is at 12. As long as thread q remains at 12, we have $X = \mathit{cnt}.q$ and, because of the assumption and (K5), also $\mathit{cnt}.r \in \{X, X + 1\}$ for all threads r. While q remains at 12, we therefore have $\mathit{old}.q = (X - 1)_R$ and $\mathtt{tag}[r] \in \{X_R, (X + 1)_R\}$ where we write x_R for x **mod** R. *We now need the additional assumption:* $R \geq 3$. Then $(X - 1)_R$, X_R, and $(X + 1)_R$ are three different numbers. It follows that thread q **at** 12 is not blocked by any of its await statements. It therefore proceeds to 10 within $N - 1$ rounds, and then within two rounds it will increment $\mathit{cnt}.q$.

Exercise 6. Show that the barrier is incorrect if we would take $R = 2$. □

Exercise 7. Assume $R > 2$ as before. The following barrier can lead to deadlock. How?

```
          var tag[0:N-1]: int := ([N] 0)

          process BarMemberB (self := 0 to N-1)
             var set := empty_set
             do true ->
10:             TNS
11:             tag[self] := (tag[self] + 1) mod R
                set := {0 .. N-1} \ {self}
```

```
12:            do is_not_empty(set) ->
                  extract some i from set
13:               await (tag[i] = tag[self])
               od
            od
         end BarMember
```

2.6 Active barriers

Barriers are often needed to allow one arbitrary thread to perform a certain action, say `Act`, like taking a snapshot of part of the global state, after which action all threads may proceed again. In that case, one could choose thread `p0` to `Act` and give all threads programs of the form

```
(3)      do true ->
            TNS
            Barrier
            if self = p0 -> Act () fi
            Barrier
         od
```

For many barriers, though not for (2), there is a better option, viz., to let `Act` be executed by the last thread at the barrier. In that case, we speak of an *active* barrier. The idea of active barriers was proposed by Arnold Meijster.

The following active barrier is even simpler than barrier (2). It uses process 0 as a coordinator and also to execute `Act`. In this way it introduces some memory contention at `tag`[0].

```
(4)       var tag[0:N-1]: bool := ([N] false)

          process BarMemberC (self := 0 to N-1)
             var cnt := 0
             do true ->
10:             TNS
11:             if self = 0 ->
                   var i := N - 1
12:                do i > 0 -> await (tag[i] != tag[self]) ; i--  od
13:                Act ()
                fi
14:             tag[self] := not tag[self] ; cnt ++
15:             await (tag[0] = tag[self])
             od
          end BarMember
```

Exercise 8. Prove correctness of barrier (4). For this purpose, you may need to prove the following invariants:

(M0) $q \textbf{ not-at } 15 \quad \Rightarrow \quad cnt.q = cnt.0$,
(M1) $cnt.0 \leq cnt.q \leq cnt.0 + 1$,
(M2) $\texttt{tag}[q] \quad \equiv \quad (cnt.q \textbf{ mod } 2 = 1)$,
(M3) $0 \textbf{ at } 12 \quad \wedge \quad i.0 < q < N \quad \Rightarrow \quad cnt.q = cnt.0 + 1$,
(M4) $0 \textbf{ in } \{13, 14\} \quad \wedge \quad 0 < q < N \quad \Rightarrow \quad cnt.q = cnt.0 + 1$.

Show that the memory contention can be eliminated by introducing an array of copies of `tag`[0].

3 Concurrency formalized

Concurrent programs are difficult to get correct. We have to be very precise in our assumptions and assertions about the programs. In this section we therefore develop a mathematical framework for the specification, the design, and the verification of such programs. In other words, it is an elaboration of section 2.3.

3.1 The basic formalism

We define a concurrent system to be a system that consists of a fixed number of processes and a (global) state. The state can only change when one of the processes takes a step. The specification of the process determines how and whether the state changes when a given process takes a step. For every process p, this specification determines a binary relation $R(p)$ between global states in the following way: $(x, y) \in R(p)$ means that, if p takes a step in state x, the resulting state can be y. We assume that every process is always able to perform a step, i.e., for every pair (p, x) there exists at least one state y with $(x, y) \in R(p)$. This assumption is called *totality.*

Remark. A process q is said to be deterministic if, for every state x, there is precisely one state y with $(x, y) \in R(q)$. The processes we explicitly implement are usually deterministic, but e.g. command `NCS` (noncritical section) is nondeterministic. Indeed, a process that executes `NCS` may choose either to remain in `NCS` or to let `NCS` terminate and go to the next command. $\square$

We write Σ for the set of possible global states; it is called the state space. Every process p is specified by the transition relation $R(p)$ on Σ. A concurrent system is specified by a state space Σ, a set *Process* of the processes, transition relations $R(p)$ for all processes p, and an initial predicate *Init*, which is a boolean function on Σ.

An execution of such a system is a sequence of pairs (x_i, p_i) with $0 \le i < m$ such that $(x_i, x_{i+1}) \in R(p_i)$ for all i with $i + 1 < m$. Informally speaking, the x_i are the subsequent states and process p_i acts in state x_i and transforms it into x_{i+1}. We do allow $m = \infty$ in which case we speak of an infinite execution. Otherwise, it is called a finite execution. A state x is called *reachable* iff it occurs in an execution that starts in an initial state, i.e., iff there is an execution such that x_0 satisfies *Init* and $x = x_i$ for some index i.

When arguing about a concurrent system, we want to avoid states and executions, and prefer to argue in terms of predicates and properties. For example, we want to define a strong invariant to be a predicate that holds initially and is preserved in every step. This is formalized in the following formal recapitulation of section 2.3.

A predicate is a boolean function on the state space Σ. If X is a predicate, we write $[X]$ to indicate that X holds everywhere, i.e., that X is identically *true*. In particular, for predicates X and Y, the assertion $[X \Rightarrow Y]$ means that Y holds in every state where X holds. We write $p : X \vartriangleright Y$ to denote that, if process p takes a step in a state where X holds, the resulting state satisfies Y. We write $X \vartriangleright Y$ (pronounced: X step Y) to indicate that $p : X \vartriangleright Y$ holds for all processes p.

Now, predicate X is defined to be a *strong invariant* iff $[\, \mathit{Init} \Rightarrow X \,]$ and $X \vartriangleright X$. Predicate X is defined to be an *invariant* iff X holds in every reachable state.

Exercise 1. Prove that every predicate implied by a strong invariant is itself invariant. Conversely, prove that every invariant is implied by some strong invariant (hint: use reachability).

Remark. The standard way to prove invariance of a predicate is therefore to construct a strong invariant that implies it. This was illustrated in the first example in section 2.3, where we also showed that not every invariant is a strong invariant.$\square$

3.2 The shared variable model

The above may sound rather abstract. In almost all applications, the processes and the global state have a much more concrete structure, given by programs and variable declarations.

The transition relation $R(p)$ of process p is usually given in the form of a sequential program that consists of several steps that are to be executed one after the other. In that case, we may number (label) the instructions and provide each process with a private variable pc to indicate its current location in the code. If $pc.p = r$, the instruction labelled r is the next one to be executed by process p. This "program counter" pc is implicitly incremented in all steps of the process, but after that it may be explicitly modified by means of a **goto** statement. Note that $R(p)$ relates the global states before and after *each single step* in the program of process p.

In practice, there are often many processes that execute the same program and have the same list of private variables. If v is a private variable, we write $v.q$ to denote the value of v of process q. A step of one process, say p, leaves the private variables of all other processes unchanged. Therefore, the extension "$.p$" is omitted in the code for process p. In the code for processes, we use the identifier *self* to denote the executing process (or its process identifier).

Usually, there are also shared variables. These can be modified by all processes. We use the convention that shared variables are written in type writer font. The global state consists of the values of all shared variables together with the values of the private variables of all processes.

If the program for a process is given by a sequential program with labels at the various locations, the command for a single location is regarded as *atomic*. Indeed, every execution is a sequential composition of such labelled commands. If the sequential program is not completely labelled in this way, we need other means to split the program in atomic commands.

We use the following convention. Atomicity of a command can be made explicit by enclosing it in the *atomicity brackets* $\langle$ and $\rangle$. In the absence of atomicity brackets, every line holds a single atomic command.

3.3 True concurrency, atomicity and non-interference

The formalism presented above is called the interleaving model, since all actions are supposed to happen instantaneously and one after the other in some nondeterminate order. In the physical reality, however, actions of different processes take time and often overlap. This is called "true concurrency". Since it is useless to analyse the overlapping of actions, when we cannot control it, we prefer to work in the interleaving model. This simplification is justified if and only if

> Every overlapping execution *that can occur* is equivalent to at least one execution in which all *actions* are interleaved; in that case the *actions* considered are called *atomic.*

Usually, the actions of the processes are determined by programming code in a language like Modula-3, SR or C. In that case, the above requirement is met by simply shared actions as defined in:

> An action is called *simply shared* if it refers at most once to a shared variable, which is of a simple type.

The concept of simply shared actions goes back to [13] (3.1).

Example. (a) Let `x` be a shared variable, let y and z be private variables, and let `w` be a shared array. Each of the following lines contains one simply shared command:

```
x := $y + 3 \cdot z$ ;
$y$ := x + $3 \cdot z$ ;
w[$y$] := $z + 1$ ;
if  x > 0  then  $y := y + 3 \cdot z$  end ;
if  $y \neq 0$  then  x := $y$  else  $y$ := w[$z$]  end  ;
```

In the last case, note that the action either refers to `x` or to `w`[z], but never to both.

(b) The incrementation `x++` of a shared variable `x` is not (guaranteed to be) a simply shared action, since it may well be implemented as `x := x+1`, which contains two references to a shared variable.

(c) The actions in the first example in section 2.3 are not simply shared. In fact, both assignments contain two references to shared variables. If one wants to rewrite the example in terms of simply shared actions, one can introduce private variables v and pc, and rewrite the programs in the form

```
A :: 0: v := n + 1 ;
     1: n := v ; goto 0 ;
B :: 2: v := n ;
     3: m := v ; goto 2 .
```

As announced above, the instruction `goto x` means `pc := x` and this assignment takes place *after* the default incrementation of pc.

(d) A shared variable `x` is called an *output* variable of thread q if q is the only thread that modifies `x`. A thread can always keep a private copy of its output variables. We therefore do not count reading of an output variable of its own as a reference to a shared variable. In section 2.6, `tag`$[q]$ is an output variable of thread q. Therefore, the commands 12, 14, and 15 of `BarMemberC` are simply shared. $\Box$

It is easy and often instructive to imagine "bigger" atomic actions. If one needs to implement them in some programming language, however, one has to take care to preclude dangerously overlapping executions. This can be achieved in two ways, either by enforcing mutual exclusion by some way of blocking, or by enforcing that overlapping executions always only affect disjoint memory locations.

3.4 Mutual exclusion and atomicity

It is often important to be able to ensure that a given command S is treated as the atomic command $\langle S \rangle$. In other words, we need a mechanism to implement the atomicity brackets $\langle$ and $\rangle$. In these lecture notes, we describe three mechanisms for this purpose: semaphores, monitors, and mutexes. All three mechanisms enforce exclusive access. For now, we concentrate on the question of exclusive access *of what*?

The setting is a number of processes in shared memory. For simplicity, we assume that all processes are executing the same program. Let R be a set of program locations. For a process q, we write q **in** R to mean that process q is at one of the locations in R, i.e., that $pc.q \in R$. Mutual exclusion for R means that there is never more than one process **in** R, i.e., that we have the invariant

(MX) $(\# q :: q \textbf{ in } R) \leq 1$.

Mutual exclusion in itself does not imply atomicity in any way.

Example. Consider a system with one shared variable `x`, initially 0, in which the processes have private variables y, also initially 0. Suppose the processes repeatedly execute the sequence of commands

```
(1)     y := x ;
        Entry ;
R :     x ++ ;   x ++ ;
        Exit ;
```

where *Entry* and *Exit* enforce mutual exclusion. So, invariant (MX) holds for the set R that consists of the two modifications of `x`. Atomicity would mean that program (1) would implement

```
(2)     y := x ;
        ⟨ x ++ ;   x ++ ⟩ .
```

Obviously, program (2) has the invariant $y \textbf{ mod } 2 = 0$, whereas this invariant is not valid for program (1). This shows that program (1) is not a valid implementation of the more abstract program (2). $\Box$

The point of the example is that the shared variable `x`, though only modified under mutual exclusion, can yet be accessed at any moment. We therefore need stronger conditions to enforce non-interference. In the example we would need to extend the set R.

We define a set R of program locations to be *abstractable* iff invariant (MX) holds, and every shared variable referred to in R is only referred to in R, and every variable referred to in R does not occur in the specification of the system. If R is abstractable, every maximal sequence of commands in R that can be subsequently executed, may be regarded as atomic. Let us call this the *abstractability principle*.

Soundness of this principle can be shown as follows. Let S be some maximal sequence of commands in R that can be subsequently executed. If we replace S by $\langle S \rangle$, this does not directly affect the specification since every variable referred to in R does not occur in the specification. It can only affect the relevant behaviour when there exist executions of the system in which the time interval of the execution of S for some process is overlapping with the time interval of some atomic command T for another process, and S and T refer to the same shared variable, say `x`. Since `x` is referred to in S, it is referred to in R and, hence, in R only. Since it is referred to in T, command T belongs to R. It follows that the two processes are concurrently in R. This is excluded by the invariant (MX).

Example. We come back to the previous example. Let us enclose $y :=$ `x` in (1) by *Entry* and *Exit*, so that now all three assignments of (1) belong to R:

(3) *Entry* ; $y :=$ `x` ; *Exit* ;
Entry ; `x ++` ; `x ++` ; *Exit* ;

System (3) violates the invariant `x` **mod** $2 = 0$, which holds for system (2). Therefore, if this invariant belongs to the system specification, system (3) still does not implement (2).

If the system specification does not refer to `x`, however, then R is abstractable and program (3) implements (2). This is argued as follows. The set R falls apart into two maximal sequences of commands. So, by the abstractability principle, it gives rise to the two atomic commands of program (2). Now that atomicity has been enforced, the four remaining commands (*Entry*, *Exit*) are equivalent to *skip* and can be omitted. Therefore, program (3) indeed implements (2).

It is not allowed to join the two maximal sequences of commands in R into one sequence, as is done in (4).

(4) $\langle\, y :=$ `x` ; `x ++` ; `x ++` $\rangle$.

Indeed, program (4) has the additional invariant $y <$ `x` which is not valid for the programs (2) and (3). □

Informally speaking, the moral of this subsection is:

Atomicity can be implied by mutual exclusion, but only if all occurrences of the relevant variables are taken care of.

We shall see the idea of abstractability again in Section 5 on pthreads. It is kept implicit in Section 4 on semaphores.

3.5 Progress

We come back to the setting of section 3.1. Progress is a general term to express that eventually a certain property occurs. A concurrent system with more than one process has many infinite executions in which not all processes participate. Such unfair executions are usually rather fruitless. We want to restrict our attention to executions in which all processes participate "enough", without imposing too severe restrictions on the order in which processes perform steps. Such restrictions are called fairness restrictions.

In the following, we describe a version of this idea which has been described as "bounded delay". If X and Y are predicates, we shall define a concept that X *leads to* Y within bounded delay. This can be interpreted to mean that, if X holds and all processes get enough opportunities to act, within bounded delay there will be a state in which Y holds.

Recall that an execution is a sequence of pairs (x_i, p_i) with $0 \leq i < m$ such that $(x_i, x_{i+1}) \in R(p.i)$ for all i with $i + 1 < m$. If $m < \infty$ and we have a second execution (y_j, q_j) with $0 \leq j < n$, and $(x_{m-1}, y_0) \in R(p_{m-1})$, then the two executions can be concatenated (glued together). Conversely, every long execution can be regarded (in several ways) as a concatenation of several shorter ones.

We define a *round* to be a finite execution, say (x_i, p_i) with $0 \leq i < m$, such that every process occurs at least once in the list $(i :: p_i)$. Note that the totality assumption of section 3.1 implies that every execution can be extended to a round.

For predicates X and Y, we define X leads to Y within bounded delay (notation $X \; o\!\!\rightarrow \; Y$) to mean that there is a number k such that every execution that is a concatenation of k rounds and that starts in a state satisfying X, also contains a state that satisfies Y.

Exercise 2. Use the definitions to prove the following rules:
(a) If $[\, X \Rightarrow Y \,]$ then $X \; o\!\!\rightarrow \; Y$.
(b) If $X \; o\!\!\rightarrow \; Y$ and $Y \; o\!\!\rightarrow \; Z$ then $X \; o\!\!\rightarrow \; Z$.
(c) If $[\, X \Rightarrow P \,]$ and $P \; o\!\!\rightarrow \; Q$ and $[\, Q \Rightarrow Y \,]$ then $X \; o\!\!\rightarrow \; Y$.
(d) If $[\, X \Rightarrow P \,]$ and $P \;\rhd\; Q$ and $[\, Q \Rightarrow Y \,]$ then $X \;\rhd\; Y$.
In the remainder of this text, we shall use these results freely.

The step operator $\rhd$ is introduced mainly because of its role in proofs of progress. It plays a key role in the progress-safety-progress rule (PSP), which in a slightly different setting goes back to [6] (page 65):

PSP-rule. Let P, Q, M, A be predicates with $P \; o\!\!\rightarrow \; Q$ and $M \wedge \neg A \;\rhd\; A \vee M$. Then we have $P \wedge M \; o\!\!\rightarrow \; A \vee (Q \wedge M)$.

Remark. The predicates P and Q are called the precondition and the postcondition, respectively. The predicates M and A are called the mainstream invariant and the alternative. $\square$

Proof. Assume that P leads to Q within k rounds. Consider an execution of k rounds that starts in a state where $P \wedge M$ holds. It suffices to prove that this execution contains a state where $A \vee (Q \wedge M)$ holds. Assume it contains no state where A holds. Since it starts in a state where M holds, the property $M \wedge \neg A \;\rhd\; A \vee M$ implies that M holds in all states of the execution. On the other hand, since P leads to Q within k rounds, the execution contains a state where Q holds. So it contains a state where $Q \wedge M$ holds. $\square$

Corollary A. Let P, Q, M be predicates with $P \, o\!\!\rightarrow \; Q$ and $M \wedge \neg Q \rhd M$. Then $P \wedge M \, o\!\!\rightarrow \; Q \wedge M$.

Proof. We use the PSP-rule with $A := M \wedge Q$. We first observe that $M \wedge \neg A = M \wedge \neg Q$ and $A \vee M = M$. Since $M \wedge \neg Q \;\rhd\; M$, we thus have $M \wedge \neg A \;\rhd\; A \vee M$. The PSP rule therefore yields $P \wedge M \; o\!\!\rightarrow \; A \vee (Q \wedge M)$, i.e., $P \wedge M \; o\!\!\rightarrow \; Q \wedge M$. $\square$

Exercise 3. Conversely, show that the PSP-rule can be derived from Corollary A.

Exercise 4. Consider a system with two shared integer variables `a` and `b`, and two processes A, B with the code

```
A :   do true  →  ⟨ if  a ≤ b  →  a ++  fi ⟩  od .
B :   do true  →  ⟨ if  b ≤ a  →  b ++  fi ⟩  od .
```

(a) Prove progress of this system in the form

$$\min(\mathtt{a}, \mathtt{b}) \geq V \quad o\!\!\rightarrow \quad \min(\mathtt{a}, \mathtt{b}) \geq W \ ,$$

for arbitrary values V and W.
(b) The atomic commands of the system are not simply shared. Implement the system with simply shared commands by means of private variables and a finer grain of atomicity.

Exercise 5. Since the definition of $o\!\!\rightarrow$ is complicated, you could propose to define $X \times\!\!\!\rightarrow Y$ if there is a constant k such that every execution (x_i, p_i) that starts in a state x_0 where X holds and in which every process occurs at least k times, also contains a state where Y holds.
(a) Show that $\times\!\!\!\rightarrow$ implies $o\!\!\rightarrow$.
(b) Use the previous exercise to show that $o\!\!\rightarrow$ does not imply $\times\!\!\!\rightarrow$.
(c) Show that $\times\!\!\!\rightarrow$ is not transitive: it does not satisfy the property analogous to exercise 2(b).

3.6 Waiting and deadlock

A process q is said to be *waiting* in state x iff it cannot alter state x, in the sense that there exists no state $y \neq x$ with $(x, y) \in R(q)$. Note that we always have $(x, x) \in R(q)$ if q is waiting in state x, since transition relation $R(q)$ is supposed to be total.

State x is said to be in *deadlock* or to have *terminated* iff in state x all processes are waiting.

Remark. We need the specification (or the intention of the program) to distinguish deadlock from termination. Termination is called deadlock when it violates the specification.

We do not speak of waiting, when process q has a nondeterminate choice to remain in state x. This is the case when both (x, x) and (x, y) are in $R(q)$ with $x \neq y$, as for example when q is executing `NCS` in a mutual exclusion problem. □

3.7 Simple and compound await statements

If B is a condition, the *simple await statement* **await** B is defined as the atomic command to wait until condition B holds. In a labelled sequential program, this command can be modelled by the atomic instruction

$$\textit{lab}: \quad \textbf{if} \ \neg B \ \rightarrow \ \textbf{goto} \ \textit{lab} \ \textbf{fi} \ .$$

Execution of this instruction when B is false, has no effect since *pc* jumps back to `lab`. If the instruction is executed when B holds, *pc* is incremented so that the next instruction can be executed next.

If B is a condition and S is a command, the *compound await statement* $\langle$ **await** B **then** S $\rangle$ is the atomic command to wait until condition B holds and then in the same action to execute S. This compound await statement can be modelled by

$$\textit{lab}: \quad \textbf{if} \ B \ \rightarrow \ S \ [\!] \ \textbf{else} \ \rightarrow \ \textbf{goto} \ \textit{lab} \ \textbf{fi} \ .$$

Await statements (of either form) have simple meanings and are convenient to use in programs. The simple await statement is also easy to implement by a so-called busy-waiting loop:

$$\textbf{do} \ \neg B \ \rightarrow \ \textit{skip} \ \textbf{od} \ ,$$

but this implementation is often undesirable since it occupies the CPU fruitlessly.

The compound await statement is harder to implement because of the required atomicity of B combined with S. Programming languages never offer the general compound await statement as a primitive since it is regarded as too costly. In this text compound await statements serve as a specification mechanism and as a stage in program development.

Exercise 6. Consider a broadcasting system with one sending process and N receiving processes that communicate by means of shared variables according to

```
var message: Message:= null

process Sender
   var b: Message
   do true ->
      b := create()
      message := b
   od end Sender
```

```
process Receiver (self := 0 to N-1)
     var b: Message
     do true ->
        b := message
        do_something_with(b)
     od end Receiver
```

This system must be synchronized by means of shared variables and simple **await** statements in such a way that *every* message of the sender is received by *all* receivers. All commands must be *simply shared*. Processes must not be forced to wait unnecessarily.

3.8 Waiting queues

The alternative to busy-waiting is to introduce a waiting queue. This can be formalized in the shared variable model by introducing a shared variable Q, which holds a set of process identifiers and which is initially empty. A process that needs to start waiting enters the set with the command *enter* Q, the semantics of which is expressed by

$$\begin{aligned}&\textit{enter } Q = \\ &\quad \langle\, Q := \; Q \cup \{\textit{self}\}\,\rangle \;;\; \langle\, \textbf{await } \textit{self} \notin Q \,\rangle \;.\end{aligned}$$

A waiting process $q \in Q$ can be freed by another process that executes $\langle\, Q := \; Q \setminus \{q\}\,\rangle$. We shall use waiting queues in the descriptions of various practical synchronization primitives below.

Note that, although we speak of *waiting queues*, the processes may be released in arbitrary order, not necessarily FIFO.

3.9 Message passing

In an important class of architectures, processes can only communicate by means of messages. There are two types of message passing: asynchronous and synchronous. In asynchronous message passing, the sender sends the message and can then perform its next action. In synchronous message passing, sending and receiving are synchronized in the sense that, after sending, the sender waits until receipt of the message.

Both types of message passing can be modelled in the shared variable model. In either case, we model messages as records with three fields: sender, destination and contents. A message passing architecture is characterized by the fact that there is only one shared variable `Mess`, which holds a bag of messages, viz. the messages that are in transit.

Process q may receive message $(s, d, c) \in$ `Mess` iff $q = d$. Reception means removal from the bag `Mess` and possibly assignment of c and s to private variables of q. If there is no message for q, then q starts waiting for a message. Some architectures, however, may allow q to turn to other activities.

In *asynchronous message passing*, the sender p sends contents c to d by simply adding the message (s, d, c) to bag `Mess`.

In *synchronous message passing*, the sender p sends contents c to d by adding the message (s, d, c) to `Mess` and then waiting until $(s, d, c) \notin$ `Mess`. Note that, in this case, `Mess` never contains more than one message sent by s. In particular, `Mess` always remains a set (rather than a general bag).

In practice, it is often convenient to introduce a buffer for each process, to hold the bag of messages in transit to this process:

$$\texttt{buf}(q) = \{(s, c) \mid (s, q, c) \in \texttt{Mess}\} \;.$$

It is then also possible to treat this buffer as a FIFO buffer, so that messages cannot pass each other in transit. We do not advise this. It restrains the implementer and it is not very useful for the applications. It is usually hard enough to argue about the bag of messages in transit, without the additional complication of the order of these messages.

4 Semaphores

The semaphore (dutch: seinpaal) is one of the oldest synchronization primitives for shared-memory systems. It was introduced by Dijkstra in 1965, see [8].

A *semaphore* `s` is a shared integer variable that determines the number of times processes may pass the synchronization point $P(\mathtt{s})$. So, a process may pass $P(\mathtt{s})$ only if $\mathtt{s} > 0$. Whenever a process passes $P(\mathtt{s})$, the value of `s` is decremented with 1. The other semaphore action is $V(\mathtt{s})$, which increments `s` with 1. The initial value of `s` is usually given at the point of declaration. The value of `s` cannot be modified or inspected by other means. Formally, we have that $P(\mathtt{s})$ and $V(\mathtt{s})$ are atomic commands given by

(1) $P(\mathtt{s}): \quad \langle\ \mathbf{await}\ \ \mathtt{s} > 0\ \ \mathbf{then}\ \ \mathtt{s} := \ s - 1\ \rangle\ ;$
$V(\mathtt{s}): \quad \langle\ \mathtt{s} := \ \mathtt{s} + 1\ \rangle\ ,$

see section 3.7 for the meaning of the compound await statement.

It follows from (1) that every semaphore `s` satisfies the semaphore invariant

(SemI) $\mathtt{s} \geq 0$.

A semaphore `s` is called a *binary* semaphore if the predicate $\mathtt{s} \leq 1$ is also invariant. Otherwise it is called a general semaphore.

4.1 Mutual exclusion with semaphores

The semaphore allows a very easy solution to the mutual exclusion problem:

```
(2)      sem s := 1  #  semaphore declaration and initialization
         process SemMutex (self := 0 to N-1)
            do true ->
0:             NCS
1:             P(s)
2:             CS
3:             V(s)
            od end SemMutex
```

Since passing `P(s)` (or `V(s)`) decrements (or increments) `s` with 1, the program text of (2) implies the invariant

(J1) $(\#\ x :: x\ \mathbf{in}\ \{2,3\}) \ = \ 1 - \mathtt{s}$.

This clearly implies the mutual exclusion property $(\#\ x :: x\ \mathbf{in}\ \{2,3\}) \leq 1$.

Exercise 1. The liveness property of (2) is that, if some process is waiting for the critical section, then within a bounded number of rounds some process will enter `CS`, formalized in

(PA) $q\ \mathbf{at}\ 1 \quad o\!\!\rightarrow \quad (\exists\ x :: x\ \mathbf{at}\ 2)$.

Prove this assertion. Also, show that the fairness property $q\ \mathbf{at}\ 1\ \ o\!\!\rightarrow\ \ q\ \mathbf{at}\ 2$ need not be valid.

4.2 Queuing semaphores

The semaphore described above is the *plain* semaphore, cf. [3].

Although the semantics of semaphores is explained in terms of a compound `await` statement, the idea is that a process waiting at `P(s)` is somehow suspended and uses no processing time until it is released by an action `V(s)`. Indeed, a slightly stronger primitive is the semaphore of [12] with a list $Q(\mathtt{s})$ of waiting processes, which are enabled again in arbitrary order. Then $P(\mathtt{s})$ and $V(\mathtt{s})$ are given by

```
(3)    $P(\mathtt{s})$ :  $\langle$ **if** $\mathtt{s} > 0$ **then** $\mathtt{s} := \mathtt{s} - 1$ **else** $Q(\mathtt{s}) := Q(\mathtt{s}) \cup \{self\}$ **fi** $\rangle$ ;
                  $\langle$ **await** $self \notin Q(\mathtt{s})$ $\rangle$ .
       $V(\mathtt{s})$ :  $\langle$ **if** $isEmpty\ (Q(\mathtt{s}))$ **then** $\mathtt{s} := \mathtt{s} + 1$
                       **else** *release one process from* $Q(\mathtt{s})$ **fi** $\rangle$ .
```

Now $P(\mathtt{s})$ is split into two atomic actions, the second of which is sometimes equivalent to *skip*. Indeed, since $Q(\mathtt{s})$ is initially empty, $P(\mathtt{s})$ only introduces waiting when $\mathtt{s} = 0$.

To see that this implements the semaphore of (1), consider an execution of a system with semaphore (3). In this execution, at some moments, some process enters queue $Q(\mathtt{s})$; at other moments a process is released from $Q(\mathtt{s})$ by a V operation. We can easily transform this execution to an execution of the system with a plain semaphore, as follows. Instead of letting a process enter $Q(\mathtt{s})$, we let it go to sleep voluntarily. Instead of releasing it from $Q(\mathtt{s})$, we let it awaken just after the V operation that would have released it.

Conversely, every execution of semaphore (1) is an execution of semaphore (3) in which queue $Q(\mathtt{s})$ remains empty. It follows that the two forms of semaphores are equivalent for all safety properties.

The semaphore of (3) is called the *queuing* semaphore. Since processes on the waiting queue form no burden for the CPU, queuing semaphores give a better performance than busy waiting, especially on single processor systems.

The queuing semaphore is stronger than the plain one. This can be shown by considering system (2) with $N = 2$ and semaphores according to (3). So, there are two processes competing for the critical section. When one process executes *CS*, while the other process is waiting, the V action of the former will enable the latter. More formally, one can prove that with two processes and queuing semaphores system (2) satisfies q **at** $1 \; o\!\!\rightarrow \; q$ **at** 2.

When doing correctness proofs with queuing semaphores, one should realize that a queuing semaphore has the invariant that, if a process is in the waiting queue, it has passed the first atomic action of action P and the semaphore has $\mathtt{s} = 0$:

$$q \in Q(\mathtt{s}) \quad \Rightarrow \quad q \ \textbf{between}\ P(\mathtt{s}) \quad \wedge \quad \mathtt{s} = 0 \ .$$

In SR, it is specified that all semaphores are queuing ones and even that the V actions release the processes in FIFO order. For correctness proofs, however, we usually assume that the semaphores are plain, since it makes the proof easier and since a program correct for plain semaphores is also correct for queuing semaphores.

4.3 An incorrect implementation of a barrier with semaphores

We may try and use semaphores to implement a barrier as discussed in section 2.5. Recall that every process is executing the loop

```
do true ->
  TNS
  Barrier
od
```

where `Barrier` is a program fragment that increments the private ghost variable *cnt*.

The problem is to implement `Barrier` such that the barrier condition (BC) holds, see 2.5. We use a shared variable `atBar` to count the processes that have reached the barrier. Recall that N is the total number of processes. We use a semaphore `mut` for mutual exclusion at the barrier and a semaphore `wait` for the processes that have to wait at the barrier. The last process that arrives at the barrier resets `atBar` and releases all waiting processes.

```
var atBar := 0 ;
sem mut := 1 ; sem wait := 0 ;
```

```
         Barrier:
11:         P(mut)
            atBar ++ ; cnt ++
            if atBar < N ->
               V(mut)  #  allow other processes to arrive!
12:            P(wait)
            [] atBar = N ->
               atBar := 0
               fa i := 1 to N-1 -> V(wait) af
               V(mut)
            fi
10:      end Barrier
```

Note that, in this code, the `P(mut)` operation is always followed by precisely one `V(mut)` operation. The argument of section 4.1 therefore implies that there is always at most one process in the region between `P(mut)` and `V(mut)`. This region may therefore be regarded as one atomic command. We can and shall therefore regard a predicate as an invariant when it holds whenever *no process* is in that region.

We have the following invariants about `atBar` and `wait`.

$$\begin{aligned}&\texttt{atBar} < N \ ,\\ &\texttt{atBar} + \texttt{wait} \quad = \quad (\# \, x :: x \ \textbf{at} \ 12) \ .\end{aligned}$$

Correctness of the barrier requires that the processes waiting at 12 proceed to 10 when the last process executes its $N-1$ statements `V(wait)`. Unfortunately, there is no guarantee that they do so. Most of the processes that are waiting at 12, will have tried to execute `P(wait)` and then have entered the queue of `wait`. In SR, these processes indeed are awakened and forced to 10. Some process, however, may have reached 12 and then fallen asleep before entering the queue of `wait`. In that case, some processes that pass the barrier may complete their next `TNS`, again reach the barrier, and pass through since `wait` is still positive. In other words, even in SR, the above barrier is incorrect.

If the probability of a context switch at any given point is 10^{-5}, testing will reveal this incorrectness only when around 10^5 processes try to pass the barrier (yet if this error had slipped into the software of an airplane, one might prefer another company).

4.4 A barrier based on a split binary semaphore

It is better first to try and solve the barrier problem in a more abstract way, viz. by means of compound await statements. Initially, the barrier is closed. Then, one after the other, each process reaches the barrier and announces its arrival by incrementing the counter `atBar`. The last process that arrives opens the barrier. While the barrier is open, the processes that pass the barrier must be disallowed to reach the barrier again before it closes. We let the barrier be closed by the last process that passes it. We thus introduce a *shared* boolean variable `open`, initially false, and we define

```
(4)      Barrier:
11:         < await not open then
              atBar ++ ; cnt ++
              if atBar = N -> open := true fi >
12:         < await open then
              atBar --
              if atBar = 0 -> open := false fi >
10:      end Barrier
```

Exercise 2. Prove that this barrier indeed satisfies the invariant

$$q \textbf{ at } 10 \quad \Rightarrow \quad cnt.q \le cnt.r \ . \ \Box$$

Inspired by barrier (4), we come back to the attempt made in section 4.3, but we now let the processes that pass the barrier decrement `atBar` one by one. In this way we obtain

```
(5)      do true ->
10:         TNS
11:         P(mut)
  b:        cnt ++ ; atBar ++
  c:        if atBar < N ->  d:  V(mut)
            [] else      ->  e:  V(wait) fi
12:         P(wait)
  b:        atBar --
  c:        if atBar > 0 ->  d:  V(wait)
            [] else      ->  e:  V(mut) fi
10:      od
```

We now have the fine-grain invariant

(Kfg) $\texttt{mut} + \texttt{wait} + \texttt{inPV} = 1$.

Here, `inPV` is the number of processes in the regions enclosed by `P..V`, i.e., in any of the locations b, c, d, e of 11 or 12.

It follows from (Kfg) that there is always at most one process in the regions enclosed by `P..V`. We may therefore regard these two regions as single atomic commands, with the numbers 1 and 2. When we do so, we have a coarser grain of atomicity, and then we have the following invariants

(K0) $\texttt{mut} + \texttt{wait} = 1$,
(K1) $\texttt{atBar} = (\# \, q :: q \textbf{ at } 12)$,
(K2) $\texttt{mut} = 1 \quad \Rightarrow \quad \texttt{atBar} < N$,
(K3) $\texttt{wait} = 1 \quad \Rightarrow \quad \texttt{atBar} > 0$.

Now the barrier invariant (BC) is implied by the invariant

(K4) $cnt.q > cnt.r \quad \equiv \quad q \textbf{ at } 12 \quad \wedge \quad \texttt{wait} = 0 \quad \wedge \quad r \textbf{ not-at } 12$.

In order to preserve (K4) when *cnt.r* is incremented, we need

(K5) $cnt.q \le cnt.r + 1$.

Preservation of (K5) when *cnt.q* is incremented follows from (K4). In this way, one proves the correctness of barrier (5).

A system of semaphores like `mut` and `wait` that work together in this way is called a *split binary semaphore.*

Exercise 3. Use the above invariants to show that there is never deadlock at the barrier.

Exercise 4. One may notice that, in (5), the last process at the barrier executes `V(wait)` and `P(wait)` in immediate succession. Transform the program by elimination of such superfluous `V` and `P` actions and of superfluous subsequent tests.

Exercise 5. Transform barrier (5) into an active barrier, cf. 2.6, and argue its correctness.

4.5 The bounded buffer

We consider a system that consists of a pair of processes that communicate via a bounded buffer. One process is called the producer. It produces items and stores them in the buffer. The other process, called the consumer, removes items from the buffer and consumes them. If the buffer is big enough, producer and consumer can proceed more or less independently. Yet, if the buffer is empty, the consumer must be blocked. If the buffer is full, the producer must be blocked.

We assume that the buffer is an array `buf` that can contain N items. We use the array as a circular buffer with indices `front` and `rear` such that `buf`[`front`] holds the first element to be removed and `buf`[`rear`] is the open position where a new element can be placed. We use an integer variable `count` to hold the number of buffer positions that are "filled".

```
var front := 0, rear := 0 , count := 0

process Producer
   do true ->
      m := produce()
      < await count < N then
          buf[rear] := m ;
          rear := (rear + 1) mod N ; count ++ >
   od end Producer

process Consumer
   do true ->
      < await count > 0 then
          m := buf[front] ;
          front := (front + 1) mod N ; count -- >
      consume (m)
   od end Consumer
```

Here, we clearly have the invariants

(L0) $0 \leq \mathtt{count} \leq N$,
(L1) $0 \leq \mathtt{front} < N$,
(L2) $\mathtt{rear} = (\mathtt{front} + \mathtt{count}) \ \mathbf{mod} \ N$.

It is now clear that Producer cannot write in a position with an unread item and that Consumer cannot read before a new item has been written.

We implement the two await statements by means of semaphores `free` and `filled`, with *free* $\approx N - \mathtt{count}$ and $\mathtt{filled} \approx \mathtt{count}$. In this way we arrive at

```
(6)       sem filled := 0, free := N

          process Producer // P
             do true ->
10:             var m := produce()
11:             P(free)
12:             buf[rear] := m
13:             < rear := (rear+1) mod N ; count ++ >
14:             V(filled)
             od end Producer

          process Consumer // C
             do true ->
20:             P(filled)
21:             var m := buf[front]
22:             < front := (front+1) mod N ; count -- >
23:             V(free)
24:             consume(m)
             od end Consumer
```

Note that `count` occurs but is not really used anymore. It can be removed from the code, but we retain it as a ghost variable, since it is useful for the proof. Since it is a ghost variable its modifications in 13 and 22 can be combined atomically with other commands.

Correctness of this program is less innocent than it may appear. For one thing, a producer and a consumer can concurrently refer to a field of `buf`. If this happens, do they refer to different fields? Also, the buffer may hold N elements. In that case, `front` equals `rear`. Is this allowed? Anyway, which elements of the buffer are in use?

It is clear that the above invariants (L1) and (L2) are still valid. The above approximation relations are sharpened to the following invariants:

(L3) $N - \texttt{free} = \texttt{count} + \#(P \textbf{ in } \{12, 13\}) + \#(C \textbf{ at } 23)$,
(L4) $\texttt{count} = \texttt{filled} + \#(P \textbf{ at } 14) + \#(C \textbf{ in } \{21, 22\})$,

where, for boolean B, we define $\#B = 1$ if B holds, and $\#B = 0$ if B is false.

The invariance of (L3) and (L4) easily follows by inspection of the commands that modify `count`, `filled`, and `free` at 11, 13, 14, 20, 22, and 23.

Since `free` and `filled` are semaphores, they have values ≥ 0. Therefore, (L3) and (L4) together imply (L0). Moreover, if P is **at** 12 and C is **at** 21, then $0 < \texttt{count} < N$ and hence $\texttt{rear} \neq \texttt{front}$, so that P and C refer to different elements of `buf`. This proves that the writing producer does not interfere with the reading consumer.

The contents of the buffer are given as the sequence

$$\textit{contents} = (\textbf{seq } i : 0 \leq i < \texttt{count} : \texttt{buf}[(\texttt{front} + i) \textbf{ mod } N]) \ .$$

This invariant is the interpretation relation: it indicates how the contents of the abstract buffer are expressed as a sequence of elements of array `buf`.

If a consumer fetches an element from the buffer (**at** 21), we have $\texttt{count} > 0$ by (L4), so the element `buf[front]` indeed belongs to the contents. In the same way, one can show that, if a producer puts an element into the buffer, it does not overwrite a current element.

Exercise 6. Assume that $N > 0$. Prove that the producer–consumer system (6) is deadlock free.

Exercise 7. Assume that there are several producers that have to fill the empty locations in the same buffer. Modify program (6) in such a way that no products get lost.

Exercise 8. Assume that there are several consumers that have to extract items from the same buffer. Modify program (6) in such a way that no items are extracted more than once.

Exercise 9. Assume that there are several producers and consumers. Can the solutions of the previous two exercises be combined?

4.6 Readers and writers

Suppose we have processes of the form

```
process Reader(i:= 1 to nr)          process Writer(i:= 1 to nw)
   do true ->                           do true ->
      NCS                                  NCS
      CSR: read data                       CSW: write data
   od end                               od end
```

The problem is that every writer must have exclusive access to the data, but that the readers must be allowed to read concurrently. Therefore, ordinary mutual exclusion is not acceptable. Formally, the safety property is captured in the required invariants

$$q \textbf{ at } \texttt{CSW} \ \wedge \ r \textbf{ at } \texttt{CSW} \ \Rightarrow \ q = r \ ,$$
$$\neg (q \textbf{ at } \texttt{CSW} \ \wedge \ r \textbf{ at } \texttt{CSR}) \ .$$

We also require that writers have priority over readers: if a writer needs to write, current readers may complete reading, but new readers must be suspended until all writers are sleeping again.

We solve the problem by first introducing three shared integer variable: `naw` for the number of active writers (at most 1), `nar` for the number of active readers, and `nww` for the number of willing (nonsleeping) writers. The solution is now expressed by refining the regions `CSR` and `CSW`:

```
(7)     CSR :
           < await nww = 0 then nar ++ >
           read data
           < nar -- >

        CSW :
           < nww ++ >
           < await naw + nar = 0 then naw ++ >
           write data
           < naw -- ; nww -- >
```

We want to implement this high-level solution by means of a split binary semaphore, where each of the atomic statements is enclosed in a `P..V` region. There are two await statements, with the guards

$$Br:\ \mathtt{nww}=0\ ,\ \text{and}\ \ Bw:\ \mathtt{naw}+\mathtt{nar}=0\ .$$

We therefore introduce binary semaphores `br` and `bw` with the invariants

$$\begin{array}{lll} \mathtt{br}=1 & \Rightarrow & Br\ \text{ and some reader is at await}\ , \\ \mathtt{bw}=1 & \Rightarrow & Bw\ \text{ and some writer is at await}\ . \end{array}$$

Since we need to determine when incrementation of `br` is useful, we introduce a shared integer variable `nwr` for the number of willing readers. We then want to have

$$\begin{array}{lll} \mathtt{br}=1 & \Rightarrow & \mathtt{nww}=0 \ \wedge\ \mathtt{nwr}>\mathtt{nar}\ , \\ \mathtt{bw}=1 & \Rightarrow & \mathtt{naw}+\mathtt{nar}=0 \ \wedge\ \mathtt{nww}>0\ . \end{array}$$

We have to increment `nwr` prior to the await statement in `CSR`. We also have three other commands that must be done under mutual exclusion. We therefore introduce a third semaphore `gs` with the split-binary-semaphore invariant

$$\mathtt{gs}+\mathtt{br}+\mathtt{bw}+\mathtt{inPV}=1\ .$$

We now implement (7) by

```
(8)     CSR :                            CSW :
           P(gs)                            P(gs)
           nwr ++                           nww ++
           VA                               VA
           P(br)                            P(bw)
           nar ++                           naw ++
           VA                               VA
           read data                        write data
           P(gs)                            P(gs)
           nar --                           naw --
           nwr --                           nww --
           VA                               VA
        end CSR                          end CSW
```

At the end of each `P..V` region, the choice which semaphore is to be opened, is determined by a new command `VA`, which is given by

```
(9)     VA =
           if naw + nar = 0 and nww > 0 -> V(bw)
           [] nww = 0 and nwr > nar -> V(br)
           [] else -> V(gs)
           fi
```

Note that, in comparison with (7), `CSR` in (8) is prepended with `< nwr ++ >`.

In the two cases where `VA` is followed immediately by a `P` action, we may distribute the `P` action over the three branches of `VA` and then replace one of the branches by *skip*. This gives clumsier but more efficient code.

4.7 General condition synchronization

In our favorite design method for concurrent algorithms, at some point the processes are synchronized by means of atomic synchronization statements that have either of two forms:

(10)
```
⟨ $S(i)$ ⟩ ,
⟨ await $B(j)$ then $T(j)$ ⟩ .
```

If nothing is known about the way in which the guards $B(\_)$ are truthified, the only conceivable implementations are forms of busy waiting. We therefore assume that all statements that may modify the variables in the guards $B(\_)$ are among the statements $S(\_)$ and $T(\_)$. In other words, the guards $B(\_)$ are only affected by the statements $S(\_)$ and $T(\_)$. It follows that a process that needs to wait for a false guard $B(\_)$, can be suspended, provided that every process that executes some statement $S(\_)$ or $T(\_)$, awakens sufficiently many waiting processes.

Following Andrews [1], p. 199, we present a method to implement such a system of related **await** statements by means of a split binary semaphore. The method is called "the technique of passing the baton".

We now show how family (10) can be implemented by means of a split binary semaphore. We use one binary semaphore `gs` for the atomic commands and one semaphore `ba`$[j]$ for each await statement. The initial values are `gs` $= 1$ and `ba`$[j] = 0$ for all j. We then have the SBS invariant

$$\texttt{inPV} + \texttt{gs} + \textstyle\sum_j \texttt{ba}[j] \quad = \quad 1 \; .$$

The atomic command ⟨ $S(i)$ ⟩ is implemented by

(11)
```
P(gs) ;   $S(i)$ ;   $VA$ .
```

We use a counter `cnt`$[j]$ to register the number of processes waiting at guard `ba`$[j]$. The await statement ⟨ **await** $B(j)$ **then** $T(j)$ ⟩ is therefore implemented by the sequence

(12)
```
P(gs) ;   cnt[$j$] ++ ;   $VA$ ;
P(ba[$j$]) ;   cnt[$j$] -- ;   $T(j)$ ;   $VA$ .
```

Command *VA* determines which part of the split binary semaphore is opened next. If j ranges from 0 to $n-1$, it is of the form

(13)
```
$VA$  =  if  cnt[0] $> 0 \wedge B(0)$  $\rightarrow$  $V$(ba[0])
         ▯  cnt[1] $> 0 \wedge B(1)$  $\rightarrow$  $V$(ba[1])
         ...
         ▯  cnt[$n-1$] $> 0 \wedge B(n-1)$  $\rightarrow$  $V$(ba[$n-1$])
         ▯  else  $\rightarrow$  $V$(gs)
         fi .
```

The **if** statement is a nondeterminate choice: any branch for which the guard `cnt`$[j] > 0 \wedge B(j)$ holds can be chosen. The programmer may strengthen these guards, but it seems preferable to choose the **else** branch only if none of the **await** branches can be chosen.

In practice, one often does not introduce arrays `cnt` and `ba` but separate variables for the array elements. Sometimes the counters `cnt` can be eliminated since there are other ways to observe waiting processes.

Since they are preceded by different statements, it is often possible to simplify the various occurrences of *VA* in different ways. Sometimes some compositions *VA* ; `P(ba`$[j]$`)` in (12) can also be simplified in a final transformation.

Example. Given is one shared boolean variable `bb` and a number of processes that need to execute

```
S :   ⟨ bb := $true$ ⟩ ,
T :   ⟨ await bb  then  bb := $false$ ⟩ .
```

The standard implementation with a split binary semaphore uses an integer variable `nq` for the number of waiting processes and two binary semaphores `gs` and `wa`. The announcement consists of `nq++`. When the process no longer waits, `nq` must be decremented. We thus get

```
var nq := 0 ;
S:    P(gs) ; bb := true ; VA0 .
T:    P(gs) ; nq++ ; VA1 ;
      P(wa) ; nq-- ; bb := false ; VA2 .
VA =  if nq > 0 and bb  ->  V(wa)
      [] else           ->  V(gs)
      fi .
```

One first takes `VA0`, `VA1` and `VA2` to be equal to `VA`. One then observes that `VA2` has the precondition $\neg$ `bb`. Therefore, `VA2` can safely be replaced by `V(gs)`. Similarly, `VA0` has the precondition `bb`. It can therefore be simplified to

```
VA0 =  if nq > 0  ->  V(wa)
       [] else    ->  V(gs)
       fi .
```

Exercise 10. Show how to simplify `VA1` in the above example. Then proceed to simplify the program for `T` in such a way that it only starts waiting if waiting is unavoidable.

Exercise 11. Show how the above technique can be used to transform program (4) into (5).

Exercise 12. Show how the above technique can be used to transform program (7) into (8) and (9).

Exercise 13. In the program for the bounded buffer of section 4.5, replace the two general semaphores `filled` and `free` by split binary semaphores in combination with some integer variables. Do not use busy waiting.

5 Posix threads

5.1 Introduction

Posix threads were introduced in 1998, when the POSIX committee specified a new standard (1003.1c) for the handling of threads and their synchronization under UNIX. Threads made according to this standard are called POSIX threads or *pthreads*.

The standard provides mutexes, condition variables and associated commands, cf. [4, 10]. In pthreads, the concepts of mutual exclusion and synchronization (which can both be realized by means of semaphores) have been separated in a nonorthogonal way. Mutexes serve for mutual exclusion and have the associated commands `lock` and `unlock`. Condition variables with the commands `wait`, `signal`, and `broadcast` serve for synchronization, but cannot be used without a mutex.

This separation is helpful. Mutexes are simpler to argue about than semaphores. Their limitations enforce a discipline in programming that makes static analysis of the code easier, by allowing a coarsening of the grain of atomicity. On the other hand, they are somewhat more flexible than Java's synchronized methods.

In subsection 5.2, we pin down the semantics of the primitives that we shall use by means of atomicity brackets and **await** statements. In subsection 5.3 we introduce the idea of mutex abstraction. This idea is worked out for three special cases in subsection 5.4. Semaphores are constructed in subsection 5.5. Subsection 5.6 contains the construction and verification of a barrier. Condition-synchronization with pthreads is presented in subsection 5.7. This theory is illustrated in subsection 5.8 by a program for logically synchronous broadcasts.

Although pthreads are programmed in C, we keep using an SR-like programming language augmented with angular brackets to express atomicity.

5.2 The semantics of mutexes and condition variables

A *mutex* is an identifier that can be used to enforce mutual exclusion by enclosing the critical sections between commands `lock` and `unlock`. It can be regarded as a special kind of semaphore that remembers the locking thread.

We only use the POSIX primitives `pthread_mutex_lock` and `pthread_mutex_unlock`, which are abbreviated here by `lock` and `unlock`. Every thread has a thread identifier, which is indicated by *self* in the code for the thread. We regard a mutex as a shared variable of type *Thread*. We say that thread p owns mutex `m` iff $\texttt{m} = p$. Mutex `m` is said to be free iff $\texttt{m} = \perp$; this holds initially. The value (owner) of a mutex can only be modified by the commands `lock` and `unlock`, which are specified by

```
(1)     lock(m) :    ⟨ await  m = ⊥  then  m := self ⟩ ;
        unlock(m) :   ⟨ assert  m = self ;   m := ⊥ ⟩ .
```

The description of `unlock` may be unexpected. It generates a runtime error when a thread tries to unlock a mutex it does not own.

A *condition variable* is an identifier to indicate the set of threads waiting for a specific condition. We therefore regard a condition variable `v` as a shared variable of the type *Set* of *Thread*, which is initially empty. We use the instruction *enter* `v` to prescribe that the acting thread places its identifier in `v` and then waits until another thread removes that identifier from `v` and thus releases the waiting thread, see section 3.8.

We only use the POSIX primitives `pthread_cond_wait`, `pthread_cond_broadcast`, `pthread_cond_signal` , and abbreviate them by `wait`, `broadcast`, and `signal`. These primitives are specified as follows.

```
(2)     wait (v, m) :
            ⟨ unlock (m); v := v ∪ {self} ⟩ ;
            ⟨ await self ∉ v ⟩ ;
            lock (m) .
```

Note that command `wait` consists of *two* atomic commands: one to start waiting and one to lock when released. Also, note that a thread must own the mutex to execute `wait`.

Command `broadcast(v)` releases all threads waiting in `v`. This implies that each of them can make progress again as soon as it acquires the mutex. This is expressed in

(3) `broadcast (v) :` $\langle$ *release all threads from* `v` $\rangle$.

Command `signal(v)` is equivalent to *skip* if `v` is empty. Otherwise, it releases at least one waiting thread. This is expressed in

(4) `signal (v) :`
$\langle$ **if** $\neg$ *isEmpty*(`v`) $\rightarrow$ *release some threads from* `v` **fi** $\rangle$.

There is one additional complication: the queues of pthread condition variables are leaky: it is allowed that a thread waiting at a condition variable `v` is released from `v` spontaneously. Such cases are called *spurious* wakeups. According to Butenhof ([5] p. 80), who was involved in the pthreads standard from its beginning, this is motivated as follows: "On some multiprocessor systems, making condition wakeup completely predictable, might substantially slow all condition variable operations. The race conditions that cause spurious wakeups should be considered rare."

The possibility of spurious wakeups implies that, for every conditional call of `wait`, the condition must be retested after awakening. Indeed, from the point of view of safety, i.e. the preservation of invariants, we may regard `wait(v,m)` as equivalent to the sequential composition of `unlock(m)` and `lock(m)` and signals and broadcast as equivalent to `skip`.

According to both [9] and [10], the `signal` may wake up more than one waiting thread. It is the intention that it usually releases only one waiting thread. We need not reckon with this, however, since it is subsumed by the spurious wakeups mentioned earlier.

It is important to realize that a thread that has been released from a condition variable `v` is not immediately running. It still has to re-acquire the mutex, in competition with other threads. Also notice that signals and broadcasts are equivalent to *skip* when `v` is empty.

Exercise 1. Consider the following (incorrect) variations of definition (2).
(a) Show that the first atomic statement of (2) must not be replaced by $\langle$ *enter* `v` ; `unlock (m)`$\rangle$.
(b) Show that the atomicity brackets of the first atomic statement of (2) must not be removed.

5.3 Mutex abstraction

In this section we formalize some of the rules of structured code locking and structured data locking [10] and describe the formal consequences of these rules. We introduce mutex guarded regions to formalize code locking, abstractability of mutexes to formalize data locking, and mutex abstraction as a program transformation to ease formal verification.

A *mutex guarded region* of mutex `m` is a code fragment of the form

(5) `lock (m) ;` S `; unlock (m)`

where S is a (possibly composite) command that does not contain `lock` or `unlock` instructions, but may contain `wait` instructions (with hidden `lock`s and `unlock`s). We write MG-region for short. MG-regions form the analogues of the PV-sections of semaphore programs and of the synchronized methods of Java.

When a thread enters an MG-region of `m`, it acquires `m`. It subsequently owns `m`, until it releases `m` by calling `unlock` or `wait`. When the thread in the MG-region executes `wait(v,m)`, it releases mutex `m`, arrives at the so-called *re-entry* location, places its thread identifier in `v`, and starts waiting. When it is released from the queue of `v`, it calls `lock(m)` again before executing the remainder of the code of S. This implies that every *active* thread in any MG-region of `m` owns the mutex. It follows that there is at most one active thread in any MG-region of `m` (mutual exclusion).

For the ease of discussion, we label the re-entry location of `wait(v,m)` with `v`$^{\#}$ where `v` is the condition variable used (we assume that, for every `v` and every thread, `wait(v,m)` occurs at most once in the code). We use the notation q **at** L to indicate that the next instruction to be executed by thread q is labelled L. Since a thread can only leave location $\mathtt{v}^{\#}$ when it has been released from `v`, we always have the invariant

$$q \in \mathtt{v} \quad \Rightarrow \quad q \ \mathbf{at}\ \mathtt{v}^{\#}\ .$$

It is useful also to introduce the set `R(v)` of the *ready* threads at the re-entry location

$$\mathtt{R(v)} = \{q \mid q\ \mathbf{at}\ \mathtt{v}^{\#}\ \wedge\ q \notin \mathtt{v}\}\ .$$

We obviously have the invariant (WL), which stands for "waiting at label":

(WL) $\quad q\ \mathbf{at}\ \mathtt{v}^{\#} \quad \equiv \quad q \in \mathtt{v} \quad \vee \quad q \in \mathtt{R(v)}\ .$

Data locking is the idea to preserve data consistency by ensuring that the data are always accessed under mutual exclusion. The idea is formalized as follows.

Definition. A shared variable is said to be *guarded* by mutex `m`, if it is referred to only in MG-regions of `m`. Mutex `m` is defined to be *abstractable* for the system if it guards every shared variable referred to in any MG-region of `m`.

If `m` is an abstractable mutex in some program *Pm*, all shared variable referred to in its MG-regions are accessed under mutual exclusion. We can therefore transform program *Pm* into a more abstract program *Pa* with a coarser grain of atomicity, such that *Pm* is guaranteed to implement *Pa*. This means that, if *Pa* satisfies some specification, then *Pm* satisfies the same specification. This transformation is called *mutex abstraction.*

The relevance of mutex abstraction is that *Pa* contains fewer atomic statements than *Pm*, so that it is usually easier to prove properties of *Pa* than of *Pm*. So, mutex abstraction helps program verification.

From the point of view of data structuring, it is natural and advisable to place all shared variables guarded by a mutex `m` together with `m` in a single structure. We leave this aside since we want to concentrate on the algorithmic aspects.

5.4 Three simple forms of mutex abstraction

It is possible to give a general recipe for the transformation of command *Pm* into *Pa*. This general recipe is quite cumbersome, since it has to reckon with the possibility that *Pm* contains a number of `wait` instructions, possibly within conditional statements and repetitions. We therefore only treat a number of relevant examples. In each case, we assume that `m` is abstractible.

The simplest case is that command S in region (5) contains no `wait` instructions. In that case mutex abstraction replaces region (5) by $\langle\, S\,\rangle$. The correctness of this transformation is shown as follows. If a thread is executing S, it holds mutex `m` and, therefore, there is no concurrent execution of an MG-region of `m`. If T is any concurrent command that does not contain MG-regions of `m`, concurrent execution of S and T is equivalent to the sequential execution $(S; T)$ (as well as to $(T; S)$), since S and T refer to disjoint sets of variables. This proves the atomicity.

We now describe mutex abstraction for an abstractable mutex `m` in the case of an MG-region that contains one `wait` instruction. Abstraction transforms the MG-region into a combination of one or two atomic commands (one starting at the beginning and possibly one starting at the re-entry location). We describe the transformation only in two special cases. In each case, we give the re-entry location $\mathtt{v}^{\#}$ and we use the label $\mathtt{m}^{\#}$ to indicate the first location after the MG-region. The command enclosed between $\mathtt{v}^{\#}$ and $\mathtt{m}^{\#}$ is called the re-entry stretch. In both cases we use S to stand for an arbitrary command that contains no `lock`, `unlock`, or `wait` instructions.

The first case is the MG-region with *repetitive initial waiting*

```
(RIW)        lock(m) ;
        v#:  do B  →  wait(v,m)  od ;
             S ;
             unlock(m) ;
```

where S is a command without `locks`, `unlocks` and `waits`. In this case, when a waiting thread has been awakened or has awakened autonomously, its next command is to retest the guard. We therefore place the re-entry label $\mathtt{v}^{\#}$ at **do**. Mutex abstraction transforms region (RIW) into the atomic command

```
(RIWt)   $\mathtt{v}^{\#}$: ⟨ **if** $B$ → *enter* v *return at* $\mathtt{v}^{\#}$
                  ⫿ **else** → $S$ **fi** ⟩ ; $\mathtt{m}^{\#}$ :
```

where the phrase *enter* `v` *return at* $\mathtt{v}^{\#}$ expresses that every suspended thread resides at location $\mathtt{v}^{\#}$ and may awaken there. Indeed, a thread that tries to execute (RIW) while B holds places its identifier in `v` and starts waiting; when it is released it has to start all over again. On the other hand, if B is false, the thread executes S and thus concludes (RIW).

A second case is the MG-region with *repetitive final waiting*:

```
(RFW)        lock(m) ;
             $S$ ;
      $\mathtt{v}^{\#}$:   do $B$ → wait(v,m) od ;
             unlock(m) ;
```

where S is a command without `locks`, `unlocks` and `waits`. In this case, a re-entrant thread needs to re-acquire the mutex only to re-test the guard. Again, the re-entry location $\mathtt{v}^{\#}$ is at **do**. Mutex abstraction transforms region (RFW) into a combination of two atomic commands

```
(RFWt)       ⟨ $S$ ; if $B$ → enter v ⫿ else → goto $\mathtt{m}^{\#}$ fi ⟩ ;
          $\mathtt{v}^{\#}$: ⟨ if $B$ → enter v return at $\mathtt{v}^{\#}$ fi ⟩ ; $\mathtt{m}^{\#}$ :
```

It is clear that MG-regions of other forms, possibly with more than one `wait` instruction, can be treated analogously. If the MG-region contains W `wait` statements, it is transformed into at most $1 + W$ atomic commands, one for entry at the beginning and one for each re-entry location, but some of these may coincide or be empty.

Exercise 2. Consider the MG-region

```
(RW1)        lock(m) ;
             $S$ ;
      $\mathtt{v}^{\#}$:   do $B$ → wait(v,m) od ;
             $T$ ;
             unlock(m) ;
```

where S and T are commands without `locks`, `unlocks` and `waits`. Apply mutex abstraction to obtain a program with two atomic commands.

5.5 Implementing semaphores

In this section, we illustrate the use of the pthread primitives and the theory of mutex abstraction by giving two implementations of semaphores as introduced in Section 4.

We first show how the thread primitives of Section 5.2 can be used to implement the plain semaphore of 4(1). For this purpose we introduce a record type with a mutex, a condition variable and a number:

```
type Sema = rec (
   m : Mutex ;
   v : ConditionVar ;
   val : int ) .
```

The field `s.val` represents the value of the abstract semaphore `s`. It must therefore be initialized with some value ≥ 0. The operation $P(\mathtt{s})$ is implemented by

```
          procedure $down$ (ref s : $Sema$)
             lock (s.m) ;
  s.v$^\#$:  do s.val $= 0 \ \rightarrow$ wait (s.v, s.m) od ;
             s.val -- ;
             unlock (s.m) ;
          end .
```

The body of *down* is an MG-region of mutex `s.m` with repetitive initial waiting (RIW) and with re-entry location `s.v`$^\#$ at **do**. The operation $V(\mathtt{s})$ is implemented by

```
          procedure $up$ (ref s : $Sema$)
             lock (s.m) ;
             s.val ++ ;
             signal (s.v) ;
             unlock (s.m) ;
          end .
```

The only shared variables referred to in *down* and *up* are the three fields of record `s`. These variables only occur in the MG-regions of *down* and *up* and are therefore guarded by mutex `s.m`. Therefore, mutex `s.m` is abstractable and we may apply mutex abstraction.

Mutex abstraction of *down* yields one atomic command of the form (RIWt), which decrements `s.val` with 1 under precondition `s.val` $\neq 0$; otherwise `s.val` is not changed.

```
  s.v$^\#$:  ⟨ if s.val $= 0 \ \rightarrow$ $enter$ s.v $return\ at$ s.v$^\#$
             ⫿ else $\rightarrow$ s.val -- fi ⟩ ; s.m$^\#$ :
```

It follows that *down* preserves the invariant

(J0) $\mathtt{s.val} \geq 0$.

The body of *up* contains no `wait` instructions, so its abstraction is a single atomic command that sends a signal and increments `s.val`. It thus also preserves (J0).

This does not yet imply correctness of the implementation. We also have to prove that threads are not waiting unnecessarily. For this purpose, we prove that, whenever some thread is enlisted in `s.v` while `s.val` > 0, there are threads ready to pass the semaphore, i.e. the set of re-entrant threads `R(s.v)` is nonempty. By contraposition, this is the same as the condition that, if `R(s.v)` is empty, `s.val` $= 0$ or `s.v` is empty. It therefore suffices to prove that after mutex abstraction the system satisfies the invariant

(EWs) $\mathtt{R(s.v)} = \emptyset \ \Rightarrow \ \mathtt{s.val} = 0 \ \vee \ \mathtt{s.v} = \emptyset$.

In order to prove that (EWs) is an invariant, we strengthen it to the invariant

(J1) $\mathtt{s.val} \leq \#\mathtt{R(s.v)} \ \vee \ \mathtt{s.v} = \emptyset$.

Indeed, it is clear that (J1) and (J0) together imply (EWs). Invariance of (J1) is shown as follows. Whenever a thread enters `s.v`, we have `s.val` $= 0$. Whenever `s.val` is incremented while `s.v` is nonempty, the set `R(s.v)` is extended by the signal in *up*. Whenever `R(s.v)` shrinks when `s.val` > 0, the field `s.val` is decremented.

One may notice that predicate (J1) is not an invariant in the setting without mutex abstraction: when a thread leaves `R(s.v)` while `s.val` > 0, it may momentarily falsify (J1) before it re-establishes (J1) by decrementing `s.val`.

In [10] p. 181, a semaphore implementation is presented that is essentially equivalent to this one; it only differs in that it uses an additional counter to avoid giving a signal when there is no waiting thread.

Notice that, if a thread performs a V operation while another thread is waiting, and then immediately performs a P operation, it may acquire the mutex first and thus pass the semaphore instead of the waiting thread. So, this indeed only implements the plain semaphore 4(1), not the queuing semaphore of 4(3).

Exercise 3. Assume that unbounded integers are available. We implement the queuing semaphore of section 4.2 by means of two shared integer variables `x` and `y`, atomicity brackets, and a simple await statement. We use `x` to count the number of calls of P and `y` to count the number of calls of V, according to the following abstract algorithm

```
P(s) : ⟨ var $t := \mathtt{x}$ ; x ++ ⟩ ;
       ⟨ await $t < \mathtt{y}$ ⟩ .
V(s) : ⟨ y ++ ⟩ .
```

Show that $s = (\mathtt{y} - \mathtt{x})$ **max** 0 represents the value of the semaphore in the sense that every call of $P(s)$ with precondition $s > 0$ decrements s and terminates, that every call of $P(s)$ with $s = 0$ leads to waiting, and that a call of $V(s)$ increments s if and only if there are no waiting threads, whereas otherwise precisely one waiting thread is released. Show that $\{p \in \mathit{Thread} \mid t.p < \mathtt{y}\}$ serves as the waiting queue.

Implement the algorithm by means of pthread primitives.

5.6 Barrier synchronization

The starting point for the implementation of a barrier with pthreads is to use a shared integer variable `atBar` to count the number of threads that have reached the barrier. Once all threads have reached the barrier, all of them can pass it. This would lead to an implementation with

```
if atBar $< N$ $\rightarrow$ wait (v, m) fi ...
```

Unfortunately, in view of the spurious wakeups for pthreads, we need to enclose the wait command in a repetition. Replacement of **if-fi** by **do-od** leads to deadlock. In order to get a test that can be applied repeatedly, we introduce a shared integer variable `tog`, initially 0, and we use a local copy of it in the barrier:

```
(6)     procedure barrier ()
          lock (m) ; atBar ++ ;
          if atBar $< N$ $\rightarrow$
            var $own$ := tog ;
$\mathtt{v}^{\#}$:      do $own$ = tog $\rightarrow$ wait (v, m) od ;
          [] else $\rightarrow$
            atBar := 0 ; tog ++ ; broadcast (v) ;
          fi ;
          unlock (m) ;
        end .
```

Exercise 4. Transform the barrier by mutex abstraction into a composition of two atomic commands.

Exercise 5. Prove the correctness of the barrier along the following lines. First, place it in an infinite loop with `TNS` and increment a private variable *cnt* next to `atbar`. Let `MinC` be the minimum of the private counters. Prove that the abstracted algorithm satisfies the invariants

(K1) $\mathtt{MinC} < \mathit{cnt}.q \;\Rightarrow\; q \textbf{ at } \mathtt{v}^{\#} \;\wedge\; \mathit{own}.q = \mathtt{tog}$,

(K2) $\mathit{cnt}.q \in \{\mathtt{MinC}, \mathtt{MinC}+1\}$.

Show that (K1) implies the barrier condition (BC) of section 2.5. Finally show that the barrier is deadlock free by means of the invariant:

$$q \textbf{ at } \mathtt{v}^{\#} \;\wedge\; \mathit{own}.q = \mathtt{tog} \;\Rightarrow\; \mathtt{MinC} < \mathit{cnt}.q \ .$$

Exercise 6. Transform barrier (6) into an active barrier.

Exercise 7. Show that the barrier (6) remains correct when the update `tog++` is replaced by $\mathtt{tog} := (\mathtt{tog} + 1)$ **mod** R for some $R \geq 2$.

Exercise 8. Show that the following variation of barrier (6) leads to deadlock:

```
procedure Barrier () =
   lock (m) ; atBar ++ ;
   if  atBar < N   →
v#:   do  atBar < N   →   wait (v,m)  od
   [] else   →
      broadcast (v) ; atBar := 0 ;
   fi ;
   unlock (m) ;
end .
```

5.7 Condition-synchronization with pthreads

Mutex abstraction is a bottom-up approach: a given program with an abstractable mutex is transformed into a more abstract program. We now turn to a top-down approach that enables us to implement condition synchronization with pthreads, as it was done with binary semaphores in section 4.7.

Assume that we have a system with a subsystem that consists of atomic commands of the form

$A(i)$: $\langle$ **await** $B(i)$ **then** $S(i)$ $\rangle$ with $0 \le i < n$.

We assume that the shared variables referred to in $S(i)$ and $B(i)$ do not occur elsewhere in the system. Usually, we also need atomic commands without any **await** conditions. These can easily be included among the commands $A(i)$ by taking $B(i) \equiv$ *true*. For convenience we place them at the end of the sequence. In other words, we assume that $B(i) \equiv$ *true* for all i with $m \le i < n$ for some $m \le n$.

5.7.1 A brute-force solution

We can implement the subsystem by means one mutex `mu` and one condition variable `cv`, and the commands `wait` and `broadcast`:

```
A_0(i) :   lock(mu) ;
           do  ¬B(i)   →   wait(cv,mu)  od ;
           S(i) ;
           if  (∃ k : B(k))   →   broadcast(cv)  fi ;
           unlock(mu) .
```

For $i \ge m$, the **do** loop can be omitted since $B(i) \equiv$ *true*.

The correctness is shown as follows. All variables referred to in any $S(i)$ or $B(i)$ are guarded by mutex `mu`. Therefore `mu` is abstractable. Mutex abstraction transforms $A_0(i)$ into the atomic command

```
cv_i#:  ⟨   if  B(i)   →
                S(i) ;
                if  (∃ k : B(k))   →   broadcast(cv)  fi ;
            [] else   →   enter cv  return at cv_i#  fi ⟩ .
```

We give the label `cv`$^\#$ the index i to indicate that thread q when waiting in $A_0(i)$ always starts testing $B(i)$. The transformation implies that $A_0(i)$ can be passed only by executing $S(i)$ under precondition $B(i)$ and precisely once. We can conceptually split the set `cv` into sets $\mathtt{cv}_i$ of the threads in `cv` that will restart at $\mathtt{cv}_i^\#$. Command `broadcast(cv)` makes all these sets empty.

We claim the invariant $q \in \mathtt{cv}_i \Rightarrow \neg B(i)$. Indeed, thread q only enters $\mathtt{cv}_i$ when $\neg B(i)$ holds. When $B(i)$ is made true, some of the variables referred to in $B(i)$ are modified. This only happens in some command $S(j)$. Since, in that case, $S(j)$ is immediately followed by `broadcast(cv)`, thread q is removed from $\texttt{cv}_i$. This proves the invariant. The invariant implies that no thread is ever waiting unnecessarily. This proves correctness of the system $A_0(i)$.

Note that the test before the broadcast can be removed since it is always allowed to give more broadcasts.

5.7.2 Using more condition variables

Since `broadcast` awakens all threads waiting at `cv`, this solution may require much communication and testing. We therefore develop an alternative in which every instance of $A(i)$ awakens at most one waiting thread, by means of a `signal` instead of a `broadcast`. Since an instance of $A(i)$ needs information on $B(j)$ to choose the `signal`, we assume that the predicates $B(j)$ only depend on shared variables. The alternative is especially attractive when the the number of waiting threads is often bigger than the number of conditions m.

So, assume that the predicates $B(i)$ only depend on shared variables. We then replace the single condition variable `cv` by an array, and also introduce an array of counters, as declared in

```
var cv [0: m-1]: ConditionVar ;
    cnt[0: m-1]: int := ([m] 0) .
```

Array `cnt` is used to avoid unnecessary generation of signals. The intention is that $\texttt{cnt}[k]$ is an upper bound of the number of threads waiting at $\texttt{cv}[k]$, as expressed by the coarse-grain invariant $\#\texttt{cv}[k] \leq \texttt{cnt}[k]$. We cannot expect an equality because of the possibility of spurious wakeups.

The await commands $A(i)$ are implemented by

```
$A_1(i)$ :     lock(mu) ;
               do $\neg B(i)$ $\rightarrow$
                  cnt[$i$] ++ ; wait(cv[$i$], mu) ;
  cv[$i$]$^\#$ :     cnt[$i$] -- ;
               od ;
               $S(i)$ ;
               $VA$ ;
               unlock(mu) .
```

In the cases with $i \geq m$, we omit the **do** loop since $B(i) \equiv true$. Hence the absence of $\texttt{cv}[i]$ and $\texttt{cnt}[i]$. Command *VA* is specified by

```
$VA$ :     if cnt[0] > 0 $\wedge$ B(0) $\rightarrow$ signal(cv[0])
           [] cnt[1] > 0 $\wedge$ B(1) $\rightarrow$ signal(cv[1])
           ...
           [] cnt[m-1] > 0 $\wedge$ B(m-1) $\rightarrow$ signal(cv[m-1])
           fi .
```

Of course, if `m` is a program variable, implementation of *VA* requires some kind of linear search.

Again, mutex `mu` is abstractable. Mutex abstraction and some further program transformation yields that the commands $A_1(i)$ can be replaced by

```
                ⟨ if $B(i)$ $\rightarrow$ $S(i)$ ; $VA$ ; goto mu$^\#$
                  [] else $\rightarrow$ cnt[$i$] ++ ; enter cv[$i$] fi ⟩ ;
  cv[$i$]$^\#$ :    ⟨ if $B(i)$ $\rightarrow$ cnt[$i$] -- ; $S(i)$ ; $VA$
                  [] else $\rightarrow$ enter cv[$i$] return at cv[$i$]$^\#$ fi ⟩ ; mu$^\#$ :
```

Again, the transformation implies that $A_1(i)$ can be passed only by executing $S(i)$ under precondition $B(i)$ and precisely once.

The signal given by *VA* is useful because of the invariant

(J0) $\texttt{cnt}[i] = \#\{q \mid q \textbf{ at } \texttt{cv}[i]^\#\}$.

Indeed, correctness of $A_1(i)$ requires that, when some thread q is waiting at $\texttt{cv}[j]$ while $B(j)$ holds, some (possibly other) thread is ready at $\texttt{cv}[k]^{\#}$ while $B(k)$ holds for some index k. This is expressed by the invariant

(J1) $\quad B(j) \ \wedge\ \texttt{cv}[j] \neq \emptyset \quad \Rightarrow \quad (\exists\, k : B(k) \ \wedge\ \texttt{R}(\texttt{cv}[k]) \neq \emptyset)$.

Predicate (J1) is preserved because of (J0) and the signal given in *VA*.

Exercise 9. Consider a system of threads with private variables p, q, r, of type integer.
(a) Use the method of section 5.7.1 to implement the abstract subsystem with shared variables `x`, `y`, `z`, of type integer, and commands of the forms

```
⟨ x := x + 1 ⟩ ,
⟨ await  x < y  then  x := x + p ⟩ ,
⟨ await  y < z  then  y := y + q ⟩ ,
⟨ await  z ≤ x  then  z := z + r ⟩ .
```

(b) Similarly using the method of section 5.7.2.

Exercise 10. Is it allowed to remove array `cnt` and replace *VA* by

VB : **if** $[\!]$ `k` :: B(`k`) $\rightarrow$ `broadcast(cv[k])` **fi** ?

If so, prove. Otherwise, provide a scenario to show what goes wrong.

Exercise 11. Transform the barrier of section 4.4 into a barrier with pthreads.

5.8 Broadcasts via a circular buffer

As an application of the theory of section 5.7, we now develop a circular buffer for broadcasts from a set of senders (producers) to a set of receivers (consumers). In other words, it is a producer-consumer system where each consumer has to consume all items produced. The concurrency problem is that the producers transfer the items to the consumers via a circular FIFO buffer of limited capacity and that every consumer must consume every an item precisely once.

5.8.1 Using compound await statements

We choose a constant $K > 1$ for the size of the buffer and declare the shared variables

```
buf[0 : K − 1] : Item ;
free : int := 0 ;
up[0 : K − 1], down[0 : K − 1] : int := ([K] 0) .
```

We use `free` for the index of the next slot for an interested producer. We use the operator $\oplus$ to denote addition modulo K. We use $\texttt{up}[i]$ as an upper estimate and $\texttt{down}[i]$ as a lower estimate of the number of consumers that yet have to read the item $\texttt{buf}[i]$. We let C stand for the number of consumers.

We give the producers q and consumers r private variables $x.q$ and $y.r$ to hold the indices of their current slots. All threads have a private variable *item* to hold an item. We introduce a shared ghost variable `Free` to number the broadcasts that have been sent and private ghost variables $Y.r$ to number the broadcasts that have been received by consumer r.

In order to preclude consumers from reading an item more than once, the producers always keep one slot empty. Before writing and incrementing `free`, they therefore have to wait until the slot at $\texttt{free} \oplus 1$ is empty, i.e. before $\texttt{up}[\texttt{free} \oplus 1] = 0$.

```
process Producer(self := 1 to P)
    var x : int := 0 ;
    do true  →
0:      produce  item ;
```

```
1:      ⟨ await up[free ⊕ 1] = 0 then
            $x$ := free ; up[$x$] := $C$ ;
            free := free ⊕ 1 ; { Free ++ } ⟩ ;
2:      buf[$x$] := $item$ ;
3:      down[$x$] := $C$ ;
    od .
```

Note that we keep the atomic coompound await statement 1 as small as possible by putting the treatment of the item in 0 and 2.

A consumer r must repeatedly wait for the slot at $y.r$ to be occupied, i.e. $\mathtt{down}[y.r] > 0$. It then can atomically decide to read the slot. We thus get

```
process Consumer(self := 1 to C)
    var $y$ : $int$ := 0 ;
    do $true$ →
4:      ⟨ await down[$y$] > 0 then down[$y$] -- ⟩ ;
5:      $item$ := buf[$y$] ;
6:      ⟨ up[$y$] -- ; $y := y \oplus 1$ ; { $Y$ ++ } ⟩ ;
7:      consume item
    od .
```

We now want to prove the safety properties that a producer never writes in a slot where a consumer is reading and that different producers always write at different slots, as expressed in

(L0) p **in** $\{2,3\} \;\wedge\; r$ **in** $\{5,6\} \;\Rightarrow\; x.p \neq y.r$,
(L1) p, q **in** $\{2,3\} \;\wedge\; x.p = x.q \;\Rightarrow\; p = q$.

In order to prove this, we postulate the invariants

(M0) $0 \leq \mathtt{down}[i]$,
(M1) $\mathtt{up}[i] \leq C$,
(M2) $\mathtt{up}[i] = C \cdot (\#\, q : q \textbf{ in } \{2,3\} \wedge x.q = i) + (\#\, r : r \textbf{ in } \{5,6\} \wedge y.r = i) + \mathtt{down}[i]$.

Indeed, since C is a positive integer, invariant (M2) together with (M0) and (M1) clearly implies that $(\#\, q : q \textbf{ in } \{2,3\} \wedge x.q = i) \leq 1$. Since this holds for all i, we get (L1). In the same way, one proves that (M2), (M0), and (M1) imply (L0).

We now show that (M0), (M1), and (M2) are invariants. All three predicates hold initially. (M0) is preserved since elements of `down` are decremented only when nonzero. (M1) is preserved since elements of `up` are only decremented or set to the value C.

Preservation of (M2) is shown as follows. When some producer p executes 1, it uses the index $x.p = \mathtt{free}$ and goes to location 2. Predicate (M2) is preserved since $\mathtt{up}[\mathtt{free}] = 0$ because of (M0) and (M2) combined with the new postulate

(M3) $\mathtt{up}[\mathtt{free}] \leq 0$.

When some producer p executes 3, (M0), (M1) and (M2) together imply that $\mathtt{down}[x.p] = 0$. Therefore, (M2) is preserved. It is easy to see that (M2) is preserved when a consumer executes 4 or 6. Preservation of (M3) follows from the code and $K > 1$.

We now want to prove that the consumers follow the producers close enough and do not rush ahead. For this purpose, we use the ghost variables `Free` and $Y.r$, and we write $Cons$ for the set of consumers. Then the requirements are captured in the invariants

(L2) $r \in Cons \;\Rightarrow\; Y.r \leq \mathtt{Free}$,
(L3) $r \in Cons \;\Rightarrow\; \mathtt{Free} - K < Y.r$.

In order to prove these invariants, we first observe the obvious invariants

(M4) $\texttt{free} = \texttt{Free}\ \mathbf{mod}\ K$,
(M5) $y.r = Y.r\ \mathbf{mod}\ K$.

We turn to the proof of (L2). Predicate (L2) is only threatened when some consumer r increments its Y in 6. (L2) is therefore preserved since (M2) implies that $\texttt{up}[y.r] > 0$, so that $y.r \neq \texttt{free}$ by (M3), and hence $Y.r \neq \texttt{Free}$ by (M4) and (M5).

Predicate (L3) is only threatened when some producer p increments `Free` in 1. So, it is preserved because of

$$Y.r = \texttt{Free} - K + 1 \quad \Rightarrow \quad \texttt{up}[y.r] \neq 0 \ ,$$

and this follows, by (M5), from the more general postulate

(M6) $\texttt{Free} - K < j < \texttt{Free} \quad \Rightarrow \quad \texttt{up}[j\ \mathbf{mod}\ K] = (\#\ r :: Y.r \leq j)$.

Preservation of (M6) when `Free` is incremented in 1 follows from (L2) and the fact that C is the number of consumers. When consumer r decrements $\texttt{up}[j\ \mathbf{mod}\ K]$ in 6, it has $y.r = j\ \mathbf{mod}\ K$ and hence $Y.r = j$ by (M4) and (M5); therefore, (M6) is also preserved in that case.

We now show that the system is deadlock free. We do this by contradiction. Assume the system is in deadlock. We then have $\texttt{up}[\texttt{free} \oplus 1] \neq 0$ and every producer p is at 1 and every consumer r is at 4 and has $\texttt{down}[y.r] = 0$. By (M6) and (L3), we have

$$\texttt{up}[\texttt{free} \oplus 1] = (\#\ r :: Y.r = \texttt{Free} - K + 1) \ .$$

Using (L2), (L3), (M4) and (M5), it follows that

$$\texttt{up}[\texttt{free} \oplus 1] = (\#\ r :: y.r = \texttt{free} \oplus 1) \ .$$

This shows that there exist consumers r with $y.r = \texttt{free} \oplus 1$. This implies $\texttt{down}[\texttt{free} \oplus 1] = 0$. By (M2), this contradicts $\texttt{up}[\texttt{free} \oplus 1] \neq 0$, thus proving that the system is deadlock free.

Exercise 12. Show that every consumer reads every item written precisely once.

5.8.2 Using pthread primitives

We now implement the atomicity brackets and the compound await statements used in the program of section 5.8.1 by means of pthread primitives. We use the method of section 5.7.1.

The shared variables `up`, `free`, and `Free` only occur in the atomic commands 1 and 6. We use the method of section 5.7.1 with a mutex `mp` and a condition variable `vp` to transform the atomic commands 1 and 6 into mutex guarded regions.

The shared variables $\texttt{down}[i]$ only occur in the atomic commands 3 with $x = i$ and 4 with $y = i$. We can therefore use the method of 5.7.1 with arrays of mutexes `mc` and condition variables `vc` to transform the atomic commands 3 and 4 into mutex guarded regions.

All ghost variables can be omitted. We claim that the program of section 5.8.1 is implemented by

```
Producers :
   do  true  →
         produce  item ;
1:       lock (mp) ;
         do  up[free ⊕ 1] ≠ 0   →   wait (vp, mp)  od ;
         x := free ;   up[x] := C ;
         free := free ⊕ 1 ;
         if  up[free ⊕ 1] = 0   →   broadcast (vp)  fi ;
         unlock (mp) ;
2:       buf[x] := item ;
3:       lock (mc[x]) ;
         down[x] := C ;
         broadcast (vc[x]) ;
         unlock (mc[x]) ;
   od .
```

```
Consumers :
   do  true   →
4:     lock (mc[$y$]) ;
       do  down[$y$] = 0   →   wait (vc[$y$], mc[$y$])  od ;
       down[$y$] -- ;
       unlock (mc[$y$]) ;
5:     $item$ := buf[$y$] ;
6:     lock (mp) ;
       up[$y$] -- ;
       $y := y \oplus 1$ ;
       if  up[free $\oplus$ 1] = 0   →   broadcast (vp)  fi ;
       unlock (mp) ;
       consume  item ;
   od  .
```

The asymmetry between mutex `mp` and the array of mutexes `mc` corresponds to the fact that all waiting producers wait at the same index, while the consumers can wait at diferent indices.

Exercise 13. Show that, in the transformation of consumer command 4, the broadcast can indeed be omitted (hint, see the argument used in 5.7.1).

6 The Java monitor

6.1 Introduction

The programming language Java offers the possibility to declare methods and blocks as *synchronized*. A synchronized method or block is almost the same as an MG-region, but every object can play the role of a mutex. Java has no condition variables. It uses commands `notify` and `notifyAll` instead of `signal` and `broadcast`.

When one thread enters a synchronized method, Java guarantees that, before it has finished, no other thread can execute any synchronized method on the same object. This mechanism therefore primarily serves to ensure mutual exclusion. (Here is one difference with pthread locking: a Java thread can pass a lock when it holds the lock itself.)

Java also offers primitives for conditional waiting. When a thread calls `wait()` inside a synchronized method of an object, it is deactivated and put into the waiting queue associated with the object. Threads waiting in the queue of an object can be awakened by calls of `notify` and `notifyAll` for this object. The call `notify()` awakens one waiting thread (if one exists), whereas `notifyAll()` awakens all threads waiting at the object.

There is one disturbing detail: the *spurious wakeups* of pthreads can also appear in Java. Indeed, since many JVM implementations are constructed using system routines as the POSIX libraries in which spurious wakeups are allowed, we have to reckon with spurious wakeups in Java as well, see Lea [11] p. 188 (footnote).

Remarks. In these notes, we only deal with concurrency, and therefore neglect all other aspects of Java programming. In particular, we do not treat the interaction between inheritance and exceptions and the primitives for concurrency. The Java mechanism is a variation of the monitor proposed by Hoare in 1974. □

6.2 A semaphore implementation in Java

In order to show that the Java primitives are at least as powerful as semaphores, we implement a semaphore in Java. A semaphore s is constructed as an object with one integer instance variable `n`, which is initialized with some value $n_0 \geq 0$. The operation $V(s)$ is implemented by

```
synchronized void up () {
    n ++ ;
    notify () ;
}
```

The operation $P(s)$ is implemented by the method

```
synchronized void down () throws InterruptedException {
    while (n == 0) wait () ;
    n -- ;
}
```

The `throws` clause is forced upon us by the compiler. This is just the Java translation of the pthread semaphore of section 5.5.

Exercise 1. Develop a barrier class in Java with a constructor that allows for a fixed number of threads. Use the ideas of section 5.6.

Exercise 2. Transform the barrier of section 4.4 into a Java barrier.

Exercise 3. Implement the broadcast system of section 5.8 in Java and test whether in simple cases all consumers reads the same items in the same order.

Exercise 4. Implement condition synchronization as described in 5.7 with Java threads. Use the method of 5.7.1.

7 General await programs

7.1 A crossing

At a crossing, traffic is coming from K different directions. It is modelled by a number of car processes that concurrently execute

```
do true ->
   Entry
   Cross
   Depart
od
```

Each car process has a private variable *dir*, its direction. To prevent accidents, it must be disallowed that cars with different directions cross at the same time. On the other hand, cars from the same direction must be allowed to cross concurrently.

We introduce a shared variable `current` for the direction currently allowed to cross. All traffic coming from directions other than `current` must be suspended. We thus want to ensure the invariant

(J0) q **at** `Cross` $\Rightarrow$ $dir.q = \texttt{current}$.

When all previously suspended traffic from the current direction has *passed* the crossing, a new current direction must be chosen, at least if there are suspended cars. When the direction has to change, the traffic on the crossing must be allowed to leave the crossing before the new direction is established.

Exercise 1. (a) Implement `Entry` and `Depart` with compound await statements, in such a way that (J0) holds and that no direction with willing cars is precluded indefinitely from crossing. When there are k cars waiting for direction d and `current` gets the new value d, `current` is not changed again before at least k cars of direction d have crossed. How do you treat the situation that there are no willing cars (e.g., deep at night)?
(b) Implement your solution with semaphores in SR.
(c) Implement your solution in Java threads.
(d) Implement your solution with pthreads.

Exercise 2. Given are two shared integer variables, `x` and `y`, initially 0, and three integer constants a, $b > 0$, and $c \geq 0$. Consider a system of several threads that occasionally need to modify `x` and `y` by commands S and T as given in:

$$S: \quad \mathtt{x} := \mathtt{x} + a \ ,$$
$$T: \quad \mathtt{y} := \mathtt{y} + b \ .$$

Synchronize the actions S and T with atomicity brackets and compound await statements in such a way that they are executed atomically and preserve the *safety* condition

(C) $\mathtt{x} \leq \mathtt{y} + c$.

The solution should be as nondeterminate as possible. Show that it does not lead to unnecessary deadlock.

Exercise 3. (from [12]) A small factory in Santa Monica makes beach chairs, umbrellas, and bikinis. The same type of wood is used in chairs and umbrellas. The same type of fabric is used in chairs, umbrellas, and bikinis. The owner of the factory wants to synchronize the activities of the three concurrent processes – process C producing chairs, process B producing bikinis, process U producing umbrellas – so as to control the inventory more precisely.

Let `f` and `w` denote the number of yards of fabric and of linear feet of wood, respectively, in the inventory. The three processes can be described as follows:

```
C :  do true -> f := f-10 ; w := w-7 ; NCS od
U :  do true -> f := f-13 ; w := w-5 ; NCS od
B :  do true -> f := f-1 ; NCS od
```

The two processes for replenishing the inventory are

```
W :  do true -> NCS ; w := w + 100 od
F :  do true -> NCS ; f := f + 45 od
```

Synchronize the activities of the five processes such that the inventory condition (J) is maintained:

(J) $\quad \mathtt{w} \geq 0 \quad \wedge \quad \mathtt{f} \geq 0 \quad \wedge \quad \mathtt{w} + \mathtt{f} \leq 500$.

(Do not worry about the dimensions in the term $\mathtt{w} + \mathtt{f}$.)

Exercise 4. After a while, the owner of the factory notices that the factory turns out bikinis only. Analysis reveals that the inventory happened to be in a state in which $\mathtt{f} > 400$. As a result thereof no wood can be added to the inventory, and the fabric supplier turns out to be eager enough to maintain $\mathtt{f} > 400$. Consequently, the production of chairs and umbrellas is suspended indefinitely. The owner of the factory therefore decides to limit the amount of fabric in stock and, for reason of symmetry, the amount of wood. The change amounts to strengthening the inventory condition (J) with individual limits on `w` and `f`:

(J) $\quad 0 \leq \mathtt{w} \leq 400 \quad \wedge \quad 0 \leq \mathtt{f} \leq 250 \quad \wedge \quad \mathtt{w} + \mathtt{f} \leq 500$.

Change the program of Exercise 3 in order to maintain the new condition.

Notice that the net effect of the stronger condition is that one of the replenishing processes may now be suspended although the factory has enough storage space available. The stronger invariant does, however, outrule the monopolistic behavior described above. It is a general rule that maximum progress (of production processes) and maximum resource utilization (of the store) do not go together.

7.2 Communication in quadruplets

Given is a number of threads and a shared array `ar[0:3]` of thread numbers. The threads are to execute the infinite repetition

```
   do true ->
10:   ncs ()
11:   Enter         12:   ...
13:   connect ()
14:   Synch         15:   ...
16:   hangup ()
17:   Exit
   od .
```

Here `ncs`, `connect`, and `hangup` are given procedures, and `connect` and `hangup` are guaranteed to terminate. The locations 12 and 15 stand for internal locations of `Enter` and `Synch`. The system must satisfy the following two conditions:

Safety. When some thread is at 13, there are exactly four threads in $\{12, 13, 14, 15\}$ and no threads in $\{16, 17\}$. Moreover, array `ar` is an enumeration of the numbers of these four threads.

Progress. When at least four threads are in $\{11, 12\}$, within a bounded number of steps a state is reached with some thread at 13. When there is a thread at 13, within a bounded number of steps a state is reached with no threads in any of the locations 12, 13, 14, 15, 16, 17.

Exercise 5. (a) Implement `Enter`, `Synch`, and `Exit` by means of atomicity brackets and await statements.
(b) Prove that your solution satisfies the requirements and that it cannot lead to deadlock.
(c) Implement the system by means of posix primitives.

7.3 The dining philosophers

A traditional synchronization problem is that of the five dining philosophers. Each of them performs an infinite alternation of thinking and eating. For thinking they need not communicate. They interact only when eating. They share a dining table on which five plates and five forks are disposed for them. Each philosopher has his own plate and there is a fork between each two adjacent plates. A philosopher needs the two forks next to his plate to eat. Therefore, no two neighbouring philosophers can be eating at the same time. This is the synchronization problem we want to solve. More precisely, we want to synchronize the eating activities of the philosophers in such a way that the following three requirements hold:

Safety: neighbours are never eating at the same time
Liveness: a hungry philosopher eventually eats
Progress: a hungry philosopher is not unnecessarily prevented from eating.

We have to assume that the act of eating necessarily terminates. Indeed, the two neighbours of a philosopher who does not stop eating, will remain hungry forever. On the other hand, thinking need not terminate: philosophers may die while thinking.

The naive proposal for a solution consists of associating a semaphore `fork`[p] with each fork. We count the fork and process numbers modulo 5 and use `right` for `self`$+1$ and `left` for `self`-1, in either case modulo 5.

```
(0)      sem fork[0:4] := ([5] 1)
         process Phil (self:= 0 to 4)
            do true ->
               think
               P(fork[self]) ; P(fork[right])
               eat
               V(fork[right]) ; V(fork[self])
            od end Phil
```

We first show that this solution is safe. It is clear that the forks are binary semaphores. When process p is eating, it is in the `P..V` regions of `fork`[p] and `fork`[$p+1$]. Therefore, the processes $p-1$ and $p+1$ are not eating at the same time. Yet, deadlock can occur: if all philosophers want to eat at the same time and each one, say p, has grabbed `fork`[p], no philosopher is able to proceed.

We now give a solution that consists of the encoding of the specification by means of await statements:

```
(1)      var ea[0:4] := ([5] false)
         process Phil (self:= 0 to 4)
            do true ->
               think
               < await not (ea[left] or ea[right]) then
                   ea[self] := true >
               eat
               ea[self] := false
            od end Phil
```

Again safety is easy. If process p is eating, `ea`[p] holds and then the processes $p-1$ and $p+1$ are forced to wait. This solution is deadlock-free, for, if a process is at the await statement with a false guard, then another process is eating and therefore can proceed. It does not satisfy the liveness requirement, however. In fact, it may happen that philosopher 1 wants to eat, but is kept hungry indefinitely, since its two neighbours 0 and 2 eat and think in alternation. This phenomenon is called individual starvation.

In order to eliminate individual starvation of philosophers, we give priority for longer waiting philophers. For this purpose, we introduce an array of booleans `active` such that `active`[p] means that p is interseted in eating. We introduce a shared integer variable `time` and private integer variables *ti* for ticket. A philopher that starts waiting, draws a ticket and increments `time`.

```
(2)      var active[0:4] := ([5] false)  #  waiting
         var time := 0

         process Phil (self:= 0 to 4)
            var ti := 0
            do true ->
10:            think
11:            active[self] := true ;
12:            < ti := time ; time ++ >
13:            await ti.self < ti.left or not active[left]
14:            await ti.self < ti.right or not active[right]
15:            eat
16:            active[self] := false
            od end Phil
```

This is against the rules: threads are not allowed to inspect private variables of other threads. A second objection is that this solution would need unbounded integers. A third objection is that we do not want to allow centralized control in the form of the shared variable `time`.

We shall meet all three objections by a later program transformation that reduces `time` and the private variables *ti.p* all to ghost variables.

Exercise 6. Prove that program (2) satisfies invariants of the form

(K0) $\texttt{active}[q] \equiv q \textbf{ in } \{\ldots\}$,
(K1) $q \textbf{ in } \{\ldots\} \wedge q+1 \textbf{ in } \{\ldots\} \Rightarrow ti.q < ti.(q+1)$,
(K2) $q \textbf{ in } \{\ldots\} \wedge q+1 \textbf{ in } \{\ldots\} \Rightarrow ti.q > ti.(q+1)$.

The invariants (K1) and (K2) together must imply safety in the form

$$\neg(q \textbf{ in } \{15,16\} \wedge q+1 \textbf{ in } \{15,16\}) .$$

Show that the philosopher with the smallest ticket is no longer active within a bounded number of rounds. Use this to prove absence of individual starvation.

Exercise 7. Introduce a shared array of booleans `last` with the invariant

(K3) $\texttt{last}[q] \equiv ti.q > ti.(q+1)$.

Modify command 12 of (2) such that (K3) is an invariant. Use (K3) to eliminate the occurrences of *ti* in the commands 13 and 14.

Remarks. Note that $\texttt{last}[x]$ is referred to only by the philosophers x and $x+1$. Also note the similarity with Peterson's algorithm.

These philosopher are short-sighted: they only communicate with their immediate neighbours. They are also rather stiff-necked: apart from action 12, they only communicate either with the lefthand neighbour or with the righthand neighbour.□

Exercise 8. Implement these dining philosophers by means of a split binary semaphore.

Exercise 9. Implement the dining philosophers by means of pthread primitives.

Exercise 10. Quarks are processes that can only execute their critical sections `CSQ` when there precisely three of them. Muons need to execute in isolation. We consider a system of $N > 3$ quarks and as many muons.

```
process Quark(self:= 1 to N)         process Muon(self:=1 to N)
   do true ->                           do true ->
      NCS                                  NCS
      CSQ                                  CSM
   od                                   od
end Quark                            end Muon
```

Quarks must only be admitted to CSQ with three of them together. Only when all three quarks have left CSQ, other processes may be allowed to enter CSQ or CSM. If there is a muon in CSM, no other process is allowed in CSQ or CSM.
(a) Control admission to CSQ and CSM by means of atomicity brackets and compound **await** statements. For this purpose, introduce counters for passing quarks. Take care that no process is kept waiting unnecessarily and that CSQ is emptied completely before new processes enter.
(b) Implement the control by means of binary semaphores.

Exercise 11. *A pipeline.* Given is a system of threads that have to pass repeatedly through a number of stages. Apart from stage 0, all stages can hold at most UPB threads. When stage 1 is empty, UPB threads from stage 0 can enter it. Stage LIM is the last stage. The threads in stage LIM can leave for stage 0 only when stage LIM is full (holds UPB threads). The threads at stage k with $0 < k <$ LIM can all pass to stage $k+1$ when stage k has been filled and all threads at $k+1$ have left. It follows that all threads at stage 1 stay together while proceeding through the pipe line. Progress is possible since gaps (empty stages) can appear at the end and move step by step to the lower stages.

Program this first with atomicity brackets and compound await statements. Prove the correctness of your solution. Implement your solution with posix primitives.

8 Message passing

In the shared variable model, every process is allowed to inspect and modify all shared variables. A good programmer will use the shared variables only, however, when that is needful for the system as a whole. In the message passing model, the latter restriction is so to speak syntactically enforced. Processes can only communicate by sending messages.

8.1 Modes of subtask delegation

In sequential programming, the only form of subtask delegation is procedure abstraction with the associated procedure calls. In concurrency, there are three other forms of subtask delegation. In shared memory systems, the main alternative of procedures consists of processes where the call is replaced by process creation (or forking). The other two forms are the rendezvous and the asynchronous message. The language SR offers these four forms of subtask delegation in a unified way as operations declared by

```
op Some_operation_name (parameter_list)
```

In the cases of a procedure and a process, the operation is implemented by a procedure body, declared in the following form:

```
proc Some_operation_name (parameter_list)
   body
end Some_operation_name
```

The difference between the procedure and the process is in the manner of activation. The two forms of activation are:

```
call Some_operation_name (arguments_list)
send Some_operation_name (arguments_list)
```

The keyword `call` (which can be omitted) gives the ordinary procedure call whereas the keyword `send` activates the proc body as a new concurrent process.

For example, the SR-declaration

```
process Oef (s := 1 to 5)
   body
end Oef
```

is shorthand for

```
op Oef (int)
proc Oef (s)
   body
end Oef
fa i := 1 to 5 -> send Oef (i) af
```

The rendezvous and the asynchronous message are also distiguished by the keywords `call` and `send`. They are, however, not implemented by associated procs (bodies), but by an input statement in another process, which is of the form

```
in Op_name_1 (parameter_list) -> body_1
[] Op_name_2 (parameter_list) -> body_2
. . .
ni
```

The input statement waits until an appropriate operation is *called* or *sent.* Then it substitutes the arguments for the parameters and executes the associated body. In the case of a **call**, the calling process is suspended until the body terminates. In the case of **send**, the calling process continues its activities without waiting. As is usual with procedures, in the body, the parameters of the operation stand for the arguments. In the case of sending, result parameters (**res**) and in–out parameters (**var**) do not make sense.

Asynchronous messages can be incorporated in the general execution model by including the bag of messages in transit in the state of the system.

Actually, since SR specifies that messages are served in fifo order, we have to be even more careful and to include all queues of messages for the various input statements in the state of the system. If we must use the fifo property in correctness proofs, the order of the queues enters the invariants. Since that gives awkward predicates, however, we shall avoid this whenever possible.

The rendezvous can be treated formally by considering it as the composition of an asynchronous message with a message back.

In the special case where the remote process expects only one message, the input statement can be replaced by the simpler receive statement. For example, if `mess` is an operation with two value parameters and the remote process has variables x and y, the following two statements are equivalent

```
in mess (a,b) -> x := a ; y := b ni
receive mess (x,y)
```

8.2 Mutual exclusion

The following solution for the mutual exclusion problem for N processes is the simplest example of the use of asynchronous messages. It uses a message `Coin`. There is only one `Coin` in the game. The process that grabs the `Coin` may enter the critical section. The system starts with the declaration and command:

```
op Coin ()
send Coin ()
```

The processes all perform the infinite loop:

```
            do true ->
0:             NCS
1:             receive Coin ()
2:             CS
3:             send Coin ()
            od .
```

Used in this way, the asynchronous message is just a semaphore: the **receive** statement is the P operation and the **send** statement is the V operation. The value of the semaphore is the number of available messages. The program starts with an isolated **send** message to initialize the "semaphore" with value 1.

To prove correctness of the above program, we need to be slightly more precise about the number of messages in transit. If `Mess` is the bag of messages in transit, we have the invariant

$$\#\texttt{Mess} + (\#x :: x \textbf{ at } \{2,3\}) = 1 \ .$$

Since $\#\texttt{Mess}$ is always ≥ 0, this implies the mutual exclusion property $(\#x :: x \textbf{ at } 2) \leq 1$.

8.3 A barrier in a ring of processes

We consider a directed ring of processes, in which the only possible communication is that a process sends asynchronous messages to its righthand neighbour and receives such messages from its lefthand neighbour. A process only "knows" the process identifiers of itself and of its two neighbours.

We now want to implement a barrier. So, all processes are in an infinite loop

$$\textbf{do}\ \ \textit{true}\ \ \rightarrow\ \ \texttt{TNS}\ ;\ \texttt{Barrier}\ \ \textbf{od}\ .$$

`TNS` is a terminating noncritical section. As usual all processes get a private ghost variable *cnt*, initially 0, which is not affected by `TNS` and which must be incremented by precisely 1 in the barrier.

The idea of the solution is that, in the barrier, the leader sends a token around which all processes that have arrived at the barrier send through. When the leader receives the token, all processes have arrived at the barrier and therefore they may pass it. In order to inform the processes of this, the token is sent around a second time.

Following a suggestion of Twan van Laarhoven (who attended the course in 2005), we initialize the structure by sending a token to the leader. In this way, all processes can use the same code and we arrive at

```
         const leader: 0 .. N-1
         op tok[0: N-1]()
         send tok[leader]()

         process Member (self := 0 to N-1)
            do true ->
10:            TNS
11:            receive tok[self]() ; send tok[right]() ; cnt ++
12:            receive tok[self]() ; send tok[right]() ;
            od
         end Member
```

We can regard 11 and 12 each as single atomic commands. Indeed, we can pretend that the message is sent immediately after the reception and that it is in transit, while actually the process is still busy with the preparations for sending (there is no observer who can distinguish these two states).

Clearly, each process p increments $cnt.p$ in the barrier by 1. For the proof of correctness, we number the processes in the ring starting at the leader, and then going to the right. So we have

$$\texttt{leader} = 0\ ,$$
$$\texttt{right}.q = (q+1)\ \textbf{mod}\ N\ .$$

We want to prove the barrier invariant

(BC) $\quad q\ \textbf{in}\ \texttt{TNS}\ \ \Rightarrow\ \ cnt.q \le cnt.r\ .$

It is easy to see that the number of messages in transit is always precisely one. So, we have

(J0) $\quad \#\ \texttt{Mess} = 1\ .$

Since we want to treat commands 11 and 12 in the same way, we use a conditional expression (as in C) to define the state functions

$$e(q) = (q\ \textbf{at}\ 12\ ?\ 1 : 0)\ ,$$
$$ecnt(q) = 2 * cnt.q - e(q)\ .$$

Whenever a process q executes command 11 or 12, it increments $ecnt(q)$ precisely with 1. This holds at 11, since q increments $cnt.q$ with 1 while the conditional expression becomes 1. It holds at 12, since the conditional expression becomes 0 again.

The tokens synchronize the processes in the following way

(J1) $\quad 0 < q < N \ \wedge\ \texttt{tok}[q] \in \texttt{Mess} \quad\Rightarrow\quad ecnt(q-1) = 1 + ecnt(q)$,
(J2) $\quad 0 < q < N \ \wedge\ ecnt(q-1) \neq ecnt(q) \quad\Rightarrow\quad \texttt{tok}[q] \in \texttt{Mess}$.

In view of (J0) this means that there is at most one $q > 0$ with $ecnt(q-1) \neq ecnt(q)$. If such q exists then $\texttt{tok}[q] \in \texttt{Mess}$ and $ecnt(q-1) = 1 + ecnt(q)$. Otherwise $\texttt{tok}[0] \in \texttt{Mess}$ holds and all values of *ecnt* are equal.

The predicates (J0) and (J1) hold initially since $\texttt{tok}[0]$ is the only message and all values of *ecnt* are 0. Every execution of 11 or 12 increments one values of *ecnt*. Execution of 11 or 12 by the leader creates an inequality in the sequence *ecnt*. Execution of 11 or 12 by q with $0 < q < N-1$ moves the inequality to the right. Execution of 11 or 12 by $N-1$ removes the inequality again.

To prove predicate (BC) we observe that, if $cnt.q > cnt.r$, then $cnt.q \geq cnt.r + 1$ and hence

$$2 * cnt.r - e(r) = ecnt(r) \geq ecnt(q) - 1 = 2 * cnt.q - e(q) - 1 \geq 2 * cnt.r - e(q) + 1 \ .$$

This implies $e(q) \geq 1 + e(r)$. Since $e(q)$ and $e(r)$ are both 0 or 1, this proves

$$cnt.q > cnt.r \quad\Rightarrow\quad q \textbf{ at } 12 \quad\wedge\quad r \textbf{ not at } 12 \ .$$

Predicate (BC) follows from this.

Progress of the barrier means that the counter of any member process get arbitrary high. Whatever the current value X of $cnt.q$ may be, it will get higher than X. This is expressed formally by

(PB) $\quad cnt.q = X \quad o\!\!\rightarrow \quad cnt.q > X$.

Exercise 1. Sketch a proof of property (PB) for the case that q is the leader (i.e. $q = 0$). Enumerate a sequence of fine-grain “leadsto” relations that together imply (PB).

8.4 Leader election in a ring

Consider a directed ring of processes as in the previous section. The processes now have to determine a leader, i.e., to set a private boolean variable *isLeader* in such a way that precisely one of the values *isLeader.q* is true.

For this purpose, we assume that every member of the ring has a private integer constant *priv* and that all constants *priv.q* are different. Let *MinP* be the minimum of the values *priv.q*.

We construct a message passing algorithm with, for each process q, the postcondition

(Q) $\quad isLeader.q \quad\equiv\quad priv.q = MinP$.

For simplicity, we assume that the processes are numbered from 0 to $N-1$, but we do not use this in the algorithm.

```
op tok[0: N-1](va: int)   # to compare values of priv
op halt[0: N-1]()         # for termination detection

process Member (self := 0 to N-1)
  var curr := priv, active := true, isLeader := false
  send tok[right](curr)
  do active ->
     in tok[self](x) ->
        if x < curr ->
           curr := x
           send tok[right](x)
        [] x = priv ->
           isLeader := true
           send halt[right]()
        fi
     [] halt[self]() ->
        active := false
        if not isLeader -> send halt[right]() fi
     ni
  od end Member
```

For the proof of the algorithm, we renumber the processes as follows. The unique process with $\mathit{priv} = \mathit{MinP}$, i.e., the future leader, gets the number $N-1$. We then number the other processes, starting with 0 at the righthand side of the future leader. As before, we have $\mathit{right}.q = (q+1)\ \mathbf{mod}\ N$. We have the obvious invariants

(K0) $\mathtt{tok}[q](x) \in \mathtt{Mess} \quad\Rightarrow\quad \mathit{MinP} \le x$,
(K1) $\mathit{curr}.(N-1) = \mathit{MinP}$.

For $q \neq N-1$, all values $\mathit{priv}.q$ are different and bigger than $\mathit{curr}.(N-1)$. We therefore have the invariants

(K2) $q \neq N-1 \quad\wedge\quad \mathtt{tok}[r](\mathit{priv}.q) \in \mathtt{Mess} \quad\Rightarrow\quad q < r$,
(K3) $q \neq N-1 \quad\Rightarrow\quad \neg\,\mathit{isLeader}.q$.

The token sent by the future leader eventually sets all values curr to MinP.

(K4) $\mathtt{tok}[r](\mathit{MinP}) \in \mathtt{Mess} \quad\wedge\quad r \le q < N \quad\Rightarrow\quad \mathit{curr}.q > \mathit{MinP}$,
(K5) $\mathtt{tok}[r](\mathit{MinP}) \in \mathtt{Mess} \quad\wedge\quad 1 \le q < r \quad\Rightarrow\quad \mathit{curr}.q = \mathit{MinP}$,
(K6) $\mathit{isLeader}.N \quad\vee\quad (\exists\, r :: \mathtt{tok}[r](\mathit{MinP}) \in \mathtt{Mess})$.

Termination detection only starts when the leader has been determined:

(K7) $\mathtt{halt}[q] \in \mathtt{Mess} \quad\Rightarrow\quad \mathit{isLeader}.N$,
(K8) $\neg\,\mathit{active}.q \quad\Rightarrow\quad \mathit{isLeader}.N$.

The processes terminate in an orderly fashion:

(K9) $\mathtt{halt}[r] \in \mathtt{Mess} \quad\Rightarrow\quad (r \le q \quad\equiv\quad \mathit{active}.q)$,
(K10) $\mathit{active}.N \quad\wedge\quad \mathit{isLeader}.N \quad\Rightarrow\quad (\exists\, r :: \mathtt{halt}[r] \in \mathtt{Mess})$.

It can then be seen that all processes terminate eventually. In the final state all messages `halt` and $\mathtt{tok}[r](\mathit{MinP})$ have been received. If the channels are not FIFO channels, it may be that some messages $\mathtt{tok}[r](x)$ remain unreceived, since they were passed by a `halt` message. It is possible to prove that this cannot happen if the channels are FIFO.

Exercise 2. (a) Show that the number of messages sent in the leader election protocol is $\mathcal{O}(N^2)$. (b) Give a scenario in which the number of messages is $\Theta(N)$ and another scenario in which it is $\Theta(N^2)$.

Exercise 3. Programming. Implement the leader election protocol and the barrier of section 8.3 in SR. The values of *priv* must be determined randomly, but all different. After electing a leader the processes must pass a number of barriers. Let `TNS` consist of a random time napping followed by writing the process number. After each `TNS` round, a newline must be written. For this purpose, extend the barrier of 8.3 to an acting barrier.

8.5 The Echo algorithm

The Echo algorithm is a graph traversal algorithm that dates back at least to 1983, cf. [15].

Given is an undirected graph that consists of a set V of nodes (vertices) and a binary relation $E \subseteq V^2$, which is symmetric and antireflexive, i.e.,

$$(x,y) \in E \quad\Rightarrow\quad (y,x) \in E \quad\wedge\quad x \neq y\ .$$

We assume that the graph is connected, i.e., the reflexive transitive closure E^* of E satisfies $E^* = V^2$.

A process is associated to each node. Processes at nodes q and r can communicate by sending messages if and only if $(q, r) \in E$. These messages are asynchronous: if q sends a message to r, this message will arrive eventually at r. The aim of the algorithm is that every node gets a parent such that, by repeatedly going from any node to its parent, one eventually arrives at some specified node.

Since this specified node has a special role, we give it a special name *starter* and we assume that it has only one edge: it is only connected with a node called *root*. Alternatively, one may regard *root* as a special node and *starter* as a virtual node added to the graph for reasons of initialization and finalization.

For each node q, the variable `nhb`$[q]$ holds the list of neighbours of q in some order. This list represents the set $\{x \mid (q, x) \in E\}$. Each node gets a private variable *lis* to hold the neighbours it has not yet received messages from. We use a procedure *Remove* to remove an element from a list, and *Pop* to fetch the first argument from a list. Nodes are represented as integers from 1 to N and *starter* $= 0$.

In this graph algorithm we use $N + 1$ different kinds of messages, one for each destination. So we use an array `token[0:N]` where the index refers to the destination. The parameters of the messages form the contents, which in this case is only the identity of the sender. So, the messages are declared by

```
op token [0: N] (sender: int)
```

The processes are declared by

```
process Node (self := 0 to N)
   var parent := self
   var lis := nhb[self]
   var p: List ; var x: int
   if self = 0 -> parent := -1 ; send token [root] (0) fi
   do lis != null ->
      in token [self] (sender) ->
         Remove (sender, lis)                 # (a)
         if parent = self ->
            parent := sender ; p := lis       # (b)
            do p != null ->
               Pop (x,p) ; send token [x] (self)  # (c)
            od
         fi
         if lis = null ->
            if self = 0 -> write ("fin")
            [] else -> send token [parent] (self) # (d)
            fi
         fi
      ni
   od end Node
```

We want to prove that the program eventually prints “fin” and that all nodes then have a parent different from themselves, such that every node is related to 0 by the transitive closure of the *parent* relation. We shall use invariants for these purposes. We let q and r range over the set of nodes $V = [0 \,.\,.\, N]$.

Before discussing invariants we need to fix the grain of atomicity. The first question is whether the `in..ni` command in the code of Node can be regarded as atomic. This is not directly obvious since it contains a loop with send actions. Yet, the messages here are asynchronous. We can therefore regard every delay of a send action as delay in transit. Since delay in transit is built into the model, we may assume that send actions are done as soon as possible. This implies that we can combine a bounded number of send actions into one atomic action. To summarize, indeed, the `in..ni` command can be regarded as atomic.

The first sanity check for a message passing algorithm is whether every message in transit will be received. This requires that $lis.q \neq \texttt{null}$ when $\texttt{token}[q]$ is in transit. We write `Mess` for the set of messages in transit. Now the sanity condition follows from the postulate

(J0) $\texttt{token}[q](s) \in \texttt{Mess} \quad \Rightarrow \quad s \in lis.q$.

In order to prove that (J0) is an invariant, we postulate

(J1) $parent.q = q \quad \wedge \quad r \in lis.q \quad \Rightarrow \quad q \in lis.r$,
(J2) $q \neq 0 \quad \wedge \quad q \neq parent.q \quad \Rightarrow \quad q \in lis.(parent.q) \quad \vee \quad lis.q = \texttt{null}$,
(J3) $(\# \; m :: m \text{ in transit from } s \text{ to } q) \leq 1$.

Indeed, (J0) holds initially since $0 \in nhb[root]$ and, initially, the only message in transit is the one from 0 to *root*. Preservation of (J0) at point (c) follows from (J1). Preservation at (a) follows from (J3). Preservation at (d) follows from (J1) and (J2). The predicates (J1), (J2), (J3) all hold initially. In order to prove preservation of them, we postulate

(J4) $\texttt{token}[d](q) \in \texttt{Mess} \quad \Rightarrow \quad parent.q \neq q$,
(J5) $\texttt{token}[parent.q](q) \in \texttt{Mess} \quad \Rightarrow \quad lis.q = \texttt{null}$,
(J6) $lis.q \subseteq nhb[q]$.

Indeed, preservation of (J1) follows from (J4); preservation of (J2) from (J0), (J1), (J5), (J6); preservation of (J3) from (J4), (J5); preservation of (J4) from (J0), (J2), (J6); preservation of (J5) from (J4); preservation of (J6) is trivial. This concludes the proof of the invariance of (J0).

Because of (J0), the sum $(\Sigma \; q :: \#lis.q)$ decreases whenever some process receive a message. This implies that the system terminates in a state with $\texttt{Mess} = \emptyset$. It is not yet clear, however, whether the outer loop of each process necessarily terminates.

The goal of the algorithm is that the transitive closure of the *parent* relation points from every node to 0. We therefore introduce the function *parent** from nodes to nodes, given by

$$\begin{array}{l} parent^*(q) \quad = \\ \quad \textbf{if} \;\; q = 0 \; \vee \; parent.q = q \;\; \textbf{then} \; q \; \textbf{else} \;\; parent^*(parent.q) \;\; \textbf{fi} \; . \end{array}$$

Initially, $parent^*(q) = q$ for all nodes q. Now it follows from (J4) that we have the invariant

(K0) $parent.q = q \quad \vee \quad parent^*(q) = 0$.

Up to this point, all invariants allow us to discard messages. This has to change: sending of messages must serve some purpose. The first invariant that shows this, is

(K1) $(q, r) \in E \quad \wedge \quad parent.q \neq q \quad \Rightarrow \quad \texttt{token}[r](q) \in \texttt{Mess} \quad \vee \quad parent.r \neq r$.

Indeed, (K1) holds initially because of the initial message from 0 to *root*. Preservation of (K1) follows from the code.

Since $parent.0 \neq 0$ and the graph is connected, predicate (K1) implies, for all nodes q:

(D0) $\texttt{Mess} = \emptyset \quad \Rightarrow \quad parent.q \neq q$.

Using (K0), we even obtain

(D1) $\texttt{Mess} = \emptyset \quad \Rightarrow \quad parent^*(q) = 0$.

We want to show that the outer loops of the processes have terminated when $\texttt{Mess} = \emptyset$. We therefore investigate the consequences of nonemptiness of *lis*. We claim the invariants

(K2) $q \in lis.r \quad \wedge \quad parent.q \notin \{q, r\} \quad \Rightarrow \quad \texttt{token}[r](q) \in \texttt{Mess}$,
(K3) $q \in lis.(parent.q) \quad \wedge \quad lis.q = \texttt{null} \quad \Rightarrow \quad \texttt{token}[parent.q](q) \in \texttt{Mess}$.

The proof of invariance of (K2) and (K3) is straight forward. We now claim

(D2) $\texttt{Mess} = \emptyset \quad \wedge \quad q \in \mathit{lis}.r \quad \Rightarrow \quad \mathit{parent}.q = r$,
(D3) $\texttt{Mess} = \emptyset \quad \wedge \quad q \in \mathit{lis}.r \quad \Rightarrow \quad \mathit{lis}.q \neq \texttt{null}$.

Indeed, (D2) follows from (K2) and (D0), and (D3) follows from (K3) and (D2).

We use these results to show, for all nodes q:

(D4) $\texttt{Mess} = \emptyset \quad \Rightarrow \quad \mathit{lis}.q = \texttt{null}$.

This goes as follows. Suppose that $\texttt{Mess} = \emptyset$. Let k_0 be a node of the graph with $\mathit{lis}.k_0 \neq \texttt{null}$. From these two assumptions we shall derive a contradiction. Given a node k_i of the graph with $\mathit{lis}.k_i \neq \texttt{null}$, we can choose $k_{i+1} \in \mathit{lis}.k_i$. Then predicate (D2) implies $\mathit{parent}.k_{i+1} = k_i$ and (D3) implies that $\mathit{lis}.k_{i+1} \neq \texttt{null}$. This yields a sequence of nodes k_i, $i \in \mathbb{N}$ in the graph with $\mathit{parent}.k_{i+1} = k_i$ for all i. Now choose $i > N$. By (D1), the *parent* path of k_i reaches node 0. Since the graph has N nodes, the path reaches 0 in N steps. Since *parent*.0 is not in the graph, this is a contradiction.

Predicate (D4) implies that when the system terminates with $\texttt{Mess} = \emptyset$, all nodes have empty *lis* and have therefore also terminated. We finally have the invariant

(K4) $\mathit{lis}.0 = \texttt{null} \quad \equiv$ "`fin`" has been printed.

So, indeed, the system does print "`fin`".

Exercise 4. Prove that "`fin`" is not printed too early:

"`fin`" has been printed $\quad \Rightarrow \quad \texttt{Mess} = \emptyset$.

Hints: you need no new invariants but several of the invariants claimed above; use (J2) to prove that $\mathit{lis}.0 = \texttt{null} \ \wedge \ \mathit{parent}^*(q) = 0$ implies $\mathit{lis}.q = \texttt{null}$.

Exercise 5. Programming. Implement the Echo algorithm in SR. Offer the user the possibility to specify the graph by giving N and the (undirected) edges. Extend the program as follows. Let every node have a value (an integer specified in the input). Use the final tokens as messages towards the starter, in such a way that the starter finally prints the sum of the values of the nodes.

8.6 Sliding window protocols

In this section we deal with a problem of fault tolerance: how to get reliable data communication via an unreliable medium. One aspect of this is the problem of error detection and error correction. Since that is a branch of applied discrete mathematics rather than concurrency, we do not deal with this and thus assume that every message either arrives correctly or is lost in transit. More precisely, we assume that the unreliability of the medium means that it may lose, duplicate, and reorder messages, but any arriving message is correct.

We model this version of fault tolerance as follows. The collection of messages is treated as a set `Mess`. Sending a message m means adding m to `Mess`. If $m \in \texttt{Mess}$ is destined for a process, that process can receive m by reading it, but m is not necessarily removed from `Mess`.

Now the problem is as follows. There is a sender process and a receiver process. The sender process has an infinite list of items, it has to communicate in a reliable way to the receiver. We formalize the list of the sender as an infinite array `a[int]`. The receiver has an infinite array `b[int]` it has to assign values to. The aim is that $\texttt{b}[i] = \texttt{a}[i]$ holds eventually for every index i.

In order to specify this, we give the receiver an integer variable *comp* with the intention that the receiver has received all item $a[i]$ with $i < \mathit{comp}$. This is captured in the required invariant

(Lq0) $0 \leq i < \mathit{comp} \quad \Rightarrow \quad \texttt{b}[i] = \texttt{a}[i]$.

Progress of the system is expressed by the requirement that *comp* grows indefinitely.

The solution is based on the idea that the sender sends repeatedly array elements together with the indices from a certain window in the array, that the receiver sends acknowledgements back to inform the sender of the initial segment that has been received correctly.

The two processes use messages from the sender to the receiver and acknowledgements back as declared by

```
op mess (i: int, v: item)
op ack (i: int)
```

A message `mess` holds a pair consisting of an index in array a together with the corresponding item. A message `ack` gives the sender an indication which items have been received. More precisely, the receiver sends the message `ack`(i) when $i = comp$. Note that `ack`(i) is not an acknowledgement of $a[i]$. These requirements are expressed in the invariants:

(Lq1) $\texttt{mess}(i, v) \in \texttt{Mess} \quad \Rightarrow \quad \texttt{a}[i] = v$,
(Lq2) $\texttt{ack}(i) \in \texttt{Mess} \quad \Rightarrow \quad i \leq comp$.

By receiving `ack`(k), the sender "knows" that all values $a[j]$ with $j < k$ have been received. It uses a private variable *down* as the upper limit of this segment. The sender does not wait for acknowledgements of single messages, but repeatedly sends the item from a window of size W starting at index *down*. The acknowledments are used to let the window slide to the right.

The sender is ready to receive acknowledgements and otherwise sends values from the window to the receiver. It thereby repeated traverses the window.

```
process Sender
   var down := 0, j := 0
   do true ->
      in ack(k) ->
         if down < k -> down := k ; j := max(j, down) fi
      [] else ->
         send mess(j, a[j])
         j := down + (j+1-down) mod W
      ni
   od
end Sender
```

It is easy to see that the sender preserves predicate (Lq1).

We assume that array b is initialized with $b[i] = \texttt{undef}$ for all indices i. Here, `undef` is a value that does not occur in array a. We decide that the receiver shall preserve the invariant

(Lq3) $b[i] = \texttt{undef} \quad \vee \quad b[i] = a[i]$.

The receiver is willing to receive a message and then to update array `b` accordingly. It checks whether its current value of *comp* can be incremented. It sends `ack` messages when *comp* is incremented and also when it has received a message that indicates that the latest `ack` may have been lost:

```
process Receiver
   var comp := 0
   do true ->
      in mess(k,x) ->
         if b[k] = undef ->
            b[k] := x
            if comp = k ->
               do b[comp] != undef -> comp ++ od
               send ack(comp)
            fi
         [] k < comp -> send ack(comp)
         fi
      ni
   od
end Receiver
```

Since the receiver only sends `ack`s with argument *comp* and only modifies *comp* by incrementation, it is easy to see that it preserves predicate (Lq2). Its modifications of array b preserve (Lq3) because of (Lq1). Its modifications of *comp* preserve (Lq0) because of (Lq3).

We do not formalize the concept of progress in this fault tolerant setting, but we can argue progress informally in the following way.

It is reasonable to assume that, when a certain kind of messages is sent often enough, eventually some of these messages will arrive. Under this assumption, we can prove that, when *comp* has a certain value, say X, it will eventually exceed X.

We distinguish two cases. First, assume that $\mathit{down} < X = \mathit{comp}$. As long as this holds, the sender will repeatedly send messages $\mathtt{mess}(k, v)$ with $k < X$. Some of these messages will arrive, prompting the receiver to send $\mathtt{ack}(X)$. Some of these messages will arrive, prompting the sender to set $\mathit{down} = X$.

The second case is $\mathit{comp} = X \leq \mathit{down}$. Here, we need two other invariants:

(Lq4a) $\quad \mathit{down} \leq \mathit{comp}$,
(Lq4b) $\quad \mathtt{b}[\mathit{comp}] = \mathtt{undef}$.

Preservation of (Lq4a) follows from (Lq2) and the code of sender. Preservation of (Lq4b) follows from the code of the receiver. Now (Lq4a) implies $\mathit{comp} = X = \mathit{down}$. As long as this holds, the sender repeatedly sends $\mathtt{mess}(k, v)$ with $k = X$. Some of these messages will arrive; because of (Lq4b) they will prompt the receiver to increment *comp* beyond X. This proves progress.

8.7 Sliding windows with bounded integers

The above version of the protocol has the disadvantage that the integers sent as array indices become arbitrary large. This leads to large messages and sometimes this is not acceptable.

Let us now assume that the medium transmits the messages in fifo order, but that it may still lose or duplicate some of them.

At this point, we notice that, in section 8.6, we also have the easily verified invariant

(Lq5) $\quad \mathtt{mess}(i, u) \in \mathtt{Mess} \quad \Rightarrow \quad i < \mathit{down} + W$.

If $\mathtt{mess}(k, v)$ is sent when $\mathtt{mess}(i, u)$ is in transit, then $\mathit{down} \leq k$, so that $i < k + W$. Therefore, if $\mathtt{mess}(i, u)$ has been sent before $\mathtt{mess}(k, v)$, then $i < k + W$. Now that the messages remain in FIFO order, it follows that, since the receiver has already received $\mathtt{mess}(\mathit{comp} - 1, b[\mathit{comp} - 1])$, we also get the invariant

(Lq6) $\quad \mathtt{mess}(k, v) \in \mathtt{Mess} \quad \Rightarrow \quad \mathit{comp} \leq k + W$.

Combining (Lq5), (Lq6), and (Lq4), we get

$$\mathit{comp} \leq k + W < \mathit{down} + 2 \cdot W \leq \mathit{comp} + 2 \cdot W \ .$$

This implies the invariant

(Lq7) $\quad \mathtt{mess}(k, v) \in \mathtt{Mess} \quad \Rightarrow \quad \mathit{comp} - W \leq k < \mathit{comp} + W$.

This implies that we can reconstruct k from k **mod** R and $\mathit{cw} = \mathit{comp} - W$, provided that $R \geq 2W$. In fact, we have

$$\begin{array}{ll} & \mathit{cw} \leq k < \mathit{cw} + R \\ \Rightarrow & \{\text{calculus}\} \\ & k - \mathit{cw} = (k - \mathit{cw}) \ \mathbf{mod}\ R \\ \Rightarrow & \{\text{a rule of arithmetic}\} \\ & k - \mathit{cw} = (k \ \mathbf{mod}\ R - \mathit{cw} \ \mathbf{mod}\ R) \ \mathbf{mod}\ R \\ \Rightarrow & \{\text{definition below}\} \\ & k = \mathit{recon}(\mathit{cw}, k \ \mathbf{mod}\ R) \ , \end{array}$$

when we define

$$recon(y, z) \quad = \quad y + (z - y \textbf{ mod } R) \textbf{ mod } R \ .$$

This formula enables Receiver to reconstruct the value of k from k **mod** R. Therefore, the sender may send k **mod** R instead of k when the receiver is modified to

```
process Receiver
   var comp := 0
   do true ->
      in mess(kk,x) ->
         var k := recon (comp - W, kk)
         if b[k] = undef ->
            b[k] := x
            if comp = k ->
               do b[comp] != undef -> comp ++ od
               send ack(comp mod R)
            fi
         [] k < comp -> send ack(comp mod R)
         fi
      ni
   od
end Receiver
```

Here, we let the argument of `ack` also be reduced modulo R.

Indeed, since Receiver only sends `ack`(*comp*) and *comp* never decreases, and *down* only increases to k when Sender recieves `ack`(k), we have the invariant

(Lq8) $\quad \texttt{ack}(k) \in \texttt{Mess} \quad \Rightarrow \quad down \leq k \leq comp \leq down + W < down + R \ .$

Therefore, Sender can reconstruct k from k **mod** R by means of the formula

$$k \quad = \quad recon(down, k \textbf{ mod } R) \ .$$

We may thus replace the sender by

```
process Sender
   var down := 0, j := 0
   do true ->
      in ack(kk) ->
         var k := recon (down, kk)
         if down < k -> down := k ; j := max(j, down) fi
      [] else ->
         send mess(j mod R, a[j])
         j := down + (j+1-down) mod W
      ni
   od
end Sender
```

In principle we could use different numbers R for the two message streams, but in practice that would be rather confusing.

Remark. The smallest instance of the protocol is the case with $W = 1$ and $R = 2$. In that case, it is called the Alternating Bit Protocol. □

Exercise 6. Programming. Implement the sliding window protocol with bounded integers. The user should be able to determine the total number of items to be transferred, the window size W and the modulus R. Model the faults by means of a probabilistic process Medium that acts as a faulty intermediary between Sender and Receiver. It loses some messages and duplicates other ones. Always test the correctness of transfer.

8.8 Logically synchronous multicasts

Let us consider a system of N processes that communicate by asynchronous messages. A *multicast* from process p to a set S of processes is a command in which p sends messages to all processes $q \in S$.

We assume that the activity of the system only consists of such multicasts. For each process, the multicasts in which it partipates, are ordered, say by its private clock. There is no global clock or global coordination, however. So, if process p participates in multicasts A and B in that order, it may well be that another process partipates in them in reverse order, or that there is a process q that participates in multicasts B and C in that order and a third process r that participates in C and A in that order.

Now the problem is to force the multicasts in a linear order. We want to assign a number (time stamp) to each multicast in such a way that, for every process p, if p participates in the multicast with number m and then in the multicast with number n, it holds that $m < n$.

For this purpose, we provide each process with a private variable *time*, which is only incremented. All messages of one multicast get the same time stamp, which equals the private time of the sender at the moment of sending. Each receiver receives its messages in the order of the time stamps. It never receives messages with time stamps above its private time. In order to prove this, each receiver gets a private ghost variable *rectime* which holds the latest time stamp received. We then have to prove that *rectime* only increases and that it always remain below *time*.

We split each process in a sender and a receiver process. Let us define a *node* to be such a combination of sender and receiver. Both sender and receiver have a private variable *time*, but the *time* of the receiver is regarded as the time of the node.

We provide each receiver process with a message queue in which the incoming messages are ordered by means of their time stamps, so that the receiving process can participate in them in the required order. Now there are two problems. Firstly, the senders must use different time stamps. Secondly, the receiver must not wait indefinitely for messages with low time stamps that will never arrive.

The first problem is easily solved by means of the following trick. We assume that the processes are numbered from 0 to $N-1$ and we decide that sender p only uses times stamps t with $t \textbf{ mod } N = p$, i.e., $t = k \cdot N + p$ for some number k.

The second problem is more serious and requires, for each multicast from p to a set S of receivers, communication between the sender p and the receivers in S. For this purpose, we introduce five kinds of messages

```
op message [0: N-1] (sender, stamp: int; va: item)
op question [0: N-1] (sender: int)
op update [0: N-1] (ti: int)
op newtime [0: N-1] (ti: int)
op answer [0: N-1] (ti: int)
```

In each case, the index gives the destination. The operation `message` serves for the actual multicast with time stamp `stamp` and contents `va`. Upon reception of `question`, the receiver sends `answer` with its current value of *time* to the sender and reserves a slot in the message queue. When the sender of a node needs to update the private time of the node, it sends a message `update` to its receiver, which is answered by means of `newtime`. The parameter `ti` stands for private time.

We use a type `intlist` to hold lists of process numbers as the sets of destinations of multicasts.

```
type intlist = ptr rec(nod: int, nx: intlist)
```

The sender processes are

```
process Sender (self := 0 to N - 1)
   var time := 0 , p: intlist
   do true ->
      p := choose () ; produce item
      iterate q over p:
         send question [q^.nod] (self)
      iterate q over p:
         in answer [self](ti) -> time:= max(time,ti) ni
      send update [self](time)
      receive newtime [self] (time)
      iterate q over p:
         send message [q^.nod] (self, time, item)
   od
end Sender
```

Procedure `choose` models that we do not know anything of the destinations that are chosen. The construction `iterate` is used as a shorthand for letting `q` range over all pointers in the list (`iterate` is not an SR primitive).

The receivers use queues to store message slots and messages that may have to wait for other messages.

```
type queue = ptr rec(ti, se: int; va: Item; nxt: queue)
```

The receiving part of every process consists of

```
      process Receiver (self := 0 to N - 1)
         var time := 0 , qu: queue := null
         do true ->
            in question [self] (sender) ->
               send answer [sender] (time)
               enqueue (time, sender, undef, qu)
            [] update [self] (ti) ->
               time := (1 + max(time, ti)/N) * N + self
               send newtime [self] (time)
            [] message [self] (sender, ti, item) ->
               time := (1 + max(time, ti)/N) * N + self
               dequeue (sender, qu)
               enqueue (time, sender, item)
               do qu != null and qu^.va != undef ->
(*)               rectime := qu^.ti
                  consume (qu^.va)
                  qu := qu^.nxt
               od
            ni
         od
      end Receiver
```

We assume that procedure `enqueue` keeps the queue ordered with increasing time stamps `ti`. In other words, the queues are priority queues. A call of `dequeue`(s, q) removes from q the (unique) triple of the form (t, s, x) with x undefined. Note that the assignments to `time` increment it always beyond `time` and `ti` and preserve the property that `time` is congruent `self` modulo N.

Correctness of the protocol is argued as follows. All messages of one multicast get the same time stamp, which equals the private time of the sending node at the moment of the decision to send.

As announced above, we also have to show that each receiver consumes its messages in the order of the time stamps and that it never consumes messages with time stamps above its private time. These two assertions are captured in the invariants

(M0) r **at** `(*)` $\Rightarrow$ $(\texttt{qu}.r)\texttt{\^{}.ti} < \mathit{rectime}$,
(M1) $\mathit{rectime}.r \leq \mathit{time}.r$,

where r ranges over all receivers. Preservation of (M0) relies on the observation that the time stamps of different multicasts are always different. Moreover, every triple with undefined item in the receiver's queue can only be replaced by a valid triple with a *higher* time stamp. It also relies on the invariants that, when receiver r is not processing a `message`, we have

(M2) $\texttt{qu}.r \neq \mathit{null} \Rightarrow (\texttt{qu}.r)\texttt{\^{}.va} = \mathit{undef}$,
(M3) x **in** $\texttt{qu}.r \Rightarrow \mathit{rectime}.r \leq x\texttt{\^{}.ti} \leq \mathit{time}.r$.

Preservation of (M1) also follows from (M3).

References

[1] Andrews, G.R.: Concurrent Programming, principles and practice. Addison-Wesley 1991.

[2] Andrews, G.R., Olsson, R.A.: The SR Programming Language, concurrency in practice. Benjamin/Cummings 1993.

[3] Apt, K.R., Olderog, E.-R.: Verification of Sequential and Concurrent Programs. Springer V. 1991.

[4] Bacon, J.: Concurrent systems. Addison Wesley Longman Ltd 1998. ISBN 0-201-17767-6.

[5] Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley 1997.

[6] Chandy, K.M., Misra, J.: Parallel Program Design, A Foundation. Addison–Wesley, 1988.

[7] Dijkstra, E.W.: The structure of the "THE"-multiprogramming system. Comm. ACM **11** (1968) 341–346.

[8] Dijkstra, E.W.: Co-operating sequential processes. In: F. Genuys (ed.): Programming Languages (NATO Advanced Study Institute). Academic Press, London etc. 1968, pp. 43–112.

[9] Harbison, S.P.: Modula–3, Prentice Hall 1992

[10] Kleiman, S., Shah, D., Smaalders, B.: Programming with Threads. SunSoft Press (Prentice Hall 1996), ISBN 0-13-172389-8.

[11] D. Lea: Concurrent Programming in Java, Design principles and patterns. Addison-Wesley 2000, ISBN 0-201-31009-0.

[12] Martin, A.J., van de Snepscheut, J.L.A.: Operating systems 1, Draft, Groningen 1988.

[13] Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs. Acta Informatica **6** (1976) 319–340.

[14] Peterson, G.L.: Myths about the mutual exclusion problem. IPL **12** (1981) 115–116.

[15] Segall, A.: Distributed network protocols. IEEE Transactions on Information Theory, IT-29 (2) 23-35, January 1983.

[16] Yih-Kuen Tsay: Deriving a scalable algorithm for mutual exclusion. In: Kutten, S. (ed.): Distributed Computing. Springer V. 1998 (LNCS 1499) pp. 394–407.

[17] Yang, J.-H., Anderson, J.H.: A fast, scalable mutual exclusion algorithm. Distr. Comput. **9** (1995) 51–60.