Alberto Pettorossi

Elements of Concurrent Programming

Third Edition

ARACNE

Table of Contents

Preface 1
1. Deterministic Computations 3
1.1 Operational Semantics of Arithmetic Expressions 3
1.2 Operational Semantics of Boolean Expressions 3
1.3 Operational Semantics of Commands 4
2. Nondeterministic Computations (Big Step Semantics) 5
2.1 Operational Semantics of Nondeterministic Commands 5
2.2 Operational Semantics of Guarded Commands 6
3. Concurrent Programs: Vectorization 7
3.1 Process Declaration and Process Call 7
3.2 Concurrency Based on Vectorization Using **fork-join** 7
3.3 Concurrency Based on Vectorization Using **cobegin-coend** 9
3.4 Example: Evaluating Recurrence Relations 10
3.5 Example: Multiplying Matrices 11
4. Concurrent Programs Based on Shared Variables 12
4.1 Preliminary Example: Prefix Sums 12
4.2 Prefix Sums Revisited 13
4.3 Semaphores 17
4.4 Semaphores and Test-and-Set Operations for Mutual Exclusion 19
4.5 Semaphores for Mutual Exclusion for Producers and Consumers 20
4.6 Private Semaphores for Establishing a Policy to Resume Processes 22
4.7 Critical Regions and Conditional Critical Regions 26
4.8 Monitors 31
4.9 Using Monitors for Solving the Five Philosophers Problem 36
4.10 Using Semaphores for Solving the Five Philosophers Problem 39
4.11 Peterson's Algorithm for Mutual Exclusion 42
4.12 Distributed Computation of Spanning Trees 51
4.13 Distributed Termination Detection 58
4.14 Lock-free and Wait-free Synchronization 63
5. Concurrent Computations in Java 64
5.1 Mutual Exclusion in Java 67
5.2 Monitors in Java 70
5.3 Bounded Buffer Monitor in Java 73
5.4 Counting Semaphore in Java 76

5.5 Binary Semaphore in Java 77
5.6 Bounded Buffer with Binary and Counting Semaphores in Java 77
5.7 Five Philosophers Problem in Java 80
5.8 Queue Monitor in Java 82
6. Concurrent Programs Based on Handshaking Communications 85
6.1 Pure CCS Calculus 85
6.2 Verifying Peterson's Algorithm for Mutual Exclusion 89
6.3 Value-Passing CCS Calculus 94
6.4 Verifying the Alternating Bit Protocol 96
7. Transactions and Serializability on Databases 101
7.1 Preliminaries 101
7.2 SerializabilityTheory 103
7.3 An Abstract View of Transactions 103
7.4 Histories and Equivalent Histories 106
7.5 The Serializability Theorem 108
7.6 Locking Protocols 109
7.7 Two Phase Locking Protocols 112
7.8 Strict Two Phase Locking Protocols 113
7.9 Deadlocks 114
7.10 Conservative Two Phase Locking Protocols 115
7. Appendix: A Distributed Program for Computing Spanning Trees .. 116
Index 129
References 132

Preface

These lecture notes are intended to introduce the reader to the basic notions of nondeterministic and concurrent programming. We start by giving the operational semantics of a simple deterministic language and the operational semantics of a simple nondeterministic language based on guarded commands. Then we consider concurrent computations based on: (i) vectorization, (ii) shared variables, and (iii) handshaking communications à la CCS (Calculus for Communicating Systems) [16]. We also address the problem of mutual exclusion and for its solution we analyze various techniques such as those based on semaphores, critical regions, conditional critical regions, and monitors. Finally, we study the problem of detecting distributed termination and the problem of the serializability of database transactions.

Sections 1, 2, and 6 are based on [16,22]. The material of Sections 3 and 4 is derived from [1,2,4,5,7,8,13,18,20]. Section 5 is based on [10] and is devoted to programming examples written in Java where the reader may see in action some of the basic techniques described in these lecture notes. In Section 7 we closely follow [3].

We would like to thank Dr. Maurizio Proietti for his many suggestions and his encouragement, Prof. Robin Milner and Prof. Matthew Hennessy for introducing me to CCS, Prof. Vijay K. Garg from whose book [10] I learnt concurrent programming in Java, my colleagues at Roma Tor Vergata University for their support and friendship, and my students for their patience and help.

Many thanks also to Dr. Gioacchino Onorati and Lorenzo Costantini of the Aracne Publishing Company for their kind and helpful cooperation.

Roma, April 2005

In the third edition we have corrected a few mistakes, we have improved Chapter 2, and we have added in the Appendix a Java program for the distributed computation of spanning trees of undirected graphs. Thanks to Dr. Emanuele De Angelis for discovering an error in the presentation of Peterson's algorithm.

Roma, January 2009

Alberto Pettorossi
Department of Informatics, Systems, and Production
University of Roma Tor Vergata
Via del Politecnico 1, I-00133 Roma, Italy
email: pettorossi@info.uniroma2.it
URL: http://www.iasi.cnr.it/~adp

1 Deterministic Computations

We begin by considering deterministic computations such as those denoted by a simple Pascal-like language. In this language we have the following syntactic categories:

- Integers

  $n, m, \ldots$ range over $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$

- Locations (or memory addresses)

  $X, Y, \ldots$ range over *Loc*

- Arithmetic Expressions

  a ranges over *AExpr* $\quad a ::= n \mid X \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2$

- Boolean Expressions

  b ranges over *BExpr* $\quad b ::= \mathit{true} \mid \mathit{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \vee b_2 \mid b_1 \wedge b_2$

- Commands

  c ranges over *Com* $\quad c ::= \mathit{skip} \mid X := a \mid c_1; c_2 \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } b \textbf{ do } c$

Below we introduce the operational semantics of our Pascal-like language. The operational semantics specifies:
- the evaluation of arithmetical expressions,
- the evaluation of boolean expressions, and
- the execution of commands.

In order to specify the operational semantics we need the notion of a state.
A *state* σ is a function from *Loc* to $\mathbb{Z}$. The set of all states is called *State*.

1.1 Operational Semantics of Arithmetic Expressions

The evaluation of arithmetic expressions is given as a subset of $\mathit{AExpr} \times \mathit{State} \times \mathbb{Z}$. A triple in $\mathit{AExpr} \times \mathit{State} \times \mathbb{Z}$ is written as $\langle a, \sigma \rangle \rightarrow n$ and specifies that the arithmetic expression a in the state σ evaluates to the integer n. The axioms and the inference rule defining the evaluation of arithmetic expressions are as follows.

$$\langle n, \sigma \rangle \rightarrow n$$

$$\langle X, \sigma \rangle \rightarrow \sigma(X)$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow n_1 \quad \langle a_2, \sigma \rangle \rightarrow n_2}{\langle a_1 \, op \, a_2, \sigma \rangle \rightarrow n_1 \, \underline{op} \, n_2}$$

where $op \in \{+, -, \times\}$ and $\underline{op}$ is the semantic operation corresponding to op.

1.2 Operational Semantics of Boolean Expressions

The evaluation of boolean expressions is given as a subset of $\mathit{BExpr} \times \mathit{State} \times \{\mathit{true}, \mathit{false}\}$. A triple in $\mathit{BExpr} \times \mathit{State} \times \{\mathit{true}, \mathit{false}\}$ is written as $\langle b, \sigma \rangle \rightarrow t$ and specifies that the boolean expression b in the state σ evaluates to the boolean value t. The axioms and the inference rules defining the evaluation of boolean expressions are as follows.

$$\langle \mathit{true}, \sigma \rangle \;\to\; \mathit{true}$$

$$\langle \mathit{false}, \sigma \rangle \;\to\; \mathit{false}$$

$$\frac{\langle a_1,\sigma\rangle \to n_1 \quad \langle a_2,\sigma\rangle \to n_2}{\langle a_1{=}a_2, \sigma\rangle \to \mathit{true}} \text{ if } n_1 = n_2 \qquad \frac{\langle a_1,\sigma\rangle \to n_1 \quad \langle a_2,\sigma\rangle \to n_2}{\langle a_1{=}a_2, \sigma\rangle \to \mathit{false}} \text{ if } n_1 \neq n_2$$

$$\frac{\langle a_1,\sigma\rangle \to n_1 \quad \langle a_2,\sigma\rangle \to n_2}{\langle a_1{\leq} a_2, \sigma\rangle \to \mathit{true}} \text{ if } n_1 \leq n_2 \qquad \frac{\langle a_1,\sigma\rangle \to n_1 \quad \langle a_2,\sigma\rangle \to n_2}{\langle a_1{\leq} a_2, \sigma\rangle \to \mathit{false}} \text{ if } n_1 \not\leq n_2$$

$$\frac{\langle b,\sigma\rangle \to \mathit{true}}{\langle \neg b,\sigma\rangle \to \mathit{false}} \qquad \frac{\langle b,\sigma\rangle \to \mathit{false}}{\langle \neg b,\sigma\rangle \to \mathit{true}}$$

$$\frac{\langle b_1,\sigma\rangle \to t_1 \quad \langle b_2,\sigma\rangle \to t_2}{\langle b_1 \; \mathit{bop} \; b_2, \sigma\rangle \to t_1 \; \underline{\mathit{bop}} \; t_2}$$

where $\mathit{bop} \in \{\vee, \wedge\}$ and $\underline{\mathit{bop}}$ is the semantic operation corresponding to bop.

1.3 Operational Semantics of Commands

We need the following notation. By $\sigma[n/X]$ we mean the function which is equal to σ except in X, where it takes the value n, that is:

$$\sigma[n/X]\,(Y) = \begin{cases} n & \text{if } Y = X \\ \sigma(Y) & \text{if } Y \neq X \end{cases}$$

The execution of commands is given as a subset of $\mathit{Com} \times \mathit{State} \times \mathit{State}$. A triple in $\mathit{Com} \times \mathit{State} \times \mathit{State}$ is written as $\langle c, \sigma\rangle \;\to\; \sigma'$ and specifies that the command c from the state σ produces the new state σ'. The axiom and the inference rules defining the execution of commands are as follows.

$$\langle \mathit{skip}, \sigma\rangle \;\to\; \sigma$$

$$\frac{\langle a,\sigma\rangle \to n}{\langle X{:=}a, \sigma\rangle \to \sigma[n/X]}$$

$$\frac{\langle c_1,\sigma\rangle \to \sigma_1 \quad \langle c_2,\sigma_1\rangle \to \sigma_2}{\langle c_1;c_2, \sigma\rangle \to \sigma_2}$$

$$\frac{\langle b,\sigma\rangle \to \mathit{true} \quad \langle c_1,\sigma\rangle \to \sigma_1}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \to \sigma_1} \qquad \frac{\langle b,\sigma\rangle \to \mathit{false} \quad \langle c_2,\sigma\rangle \to \sigma_2}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \to \sigma_2}$$

$$\frac{\langle b,\sigma\rangle \to \mathit{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma} \qquad \frac{\langle b,\sigma\rangle \to \mathit{true} \quad \langle c,\sigma\rangle \to \sigma_1 \quad \langle \textbf{while } b \textbf{ do } c, \sigma_1\rangle \to \sigma_*}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \to \sigma_*}$$

Note 1. The ternary operator $\langle \_, \_\rangle \to \_$ is overloaded and it is used for the operational semantics of arithmetic expressions, boolean expressions, and commands.

Here is Euclid's algorithm for computing the greatest common divisor of M and N using commands (we assume that $M > 0$ and $N > 0$):

$\{n = N > 0 \wedge m = M > 0\}$

while $m \neq n$ **do if** $m > n$ **then** $m := m{-}n$ **else** $n := n{-}m$ **od**

$\{gcd(N, M) \;=\; n\}$

2 Nondeterministic Computations (Big Step Semantics)

We present the guarded commands first introduced by Dijkstra [9]. They allow nondeterministic computations. We have the following new syntactic categories (which are mutually recursively defined):

- Commands
  c ranges over Com $\qquad c ::= skip \mid abort \mid X := a \mid c_1; c_2 \mid \textbf{if } gc \textbf{ fi} \mid \textbf{do } gc \textbf{ od}$
- Guarded Commands
  gc ranges over $GCom$ $\qquad gc ::= b \rightarrow c \mid gc_1 [\!] gc_2$

where a ranges over the arithmetic expressions $AExpr$ and b ranges over the boolean expressions $BExpr$. The operator $[\!]$ is assumed to be associative and commutative. The operational semantics given below specifies:
- the execution of commands and
- the execution of guarded commands.

2.1 Operational Semantics of Nondeterministic Commands

The execution of commands is given as a subset of $Com \times State \times State$. A triple in $Com \times State \times State$ is written as $\langle c, \sigma\rangle \rightarrow \gamma$ and specifies that the command c from the state σ produces the new state γ. The axiom and the inference rules defining the execution of commands are as follows.

(1.1) $\langle skip, \sigma\rangle \rightarrow \sigma$

(1.2) $$\frac{\langle a,\sigma\rangle \rightarrow n}{\langle X{:=}a, \sigma\rangle \rightarrow \sigma[n/X]}$$

(1.3) $$\frac{\langle c_1,\sigma\rangle \rightarrow \sigma_1 \qquad \langle c_2,\sigma_1\rangle \rightarrow \sigma_2}{\langle c_1; c_2, \sigma\rangle \rightarrow \sigma_2}$$

(1.4) $$\frac{\langle gc,\sigma\rangle \rightarrow \langle c,\sigma\rangle \qquad \langle c,\sigma\rangle \rightarrow \sigma'}{\langle \textbf{if } gc \textbf{ fi}, \sigma\rangle \rightarrow \sigma'}$$

(1.5) $$\frac{\langle gc,\sigma\rangle \rightarrow \textbf{fail}}{\langle \textbf{do } gc \textbf{ od}, \sigma\rangle \rightarrow \sigma}$$

(1.6) $$\frac{\langle gc,\sigma\rangle \rightarrow \langle c,\sigma\rangle \qquad \langle c; \textbf{do } gc \textbf{ od}, \sigma\rangle \rightarrow \sigma'}{\langle \textbf{do } gc \textbf{ od}, \sigma\rangle \rightarrow \sigma'}$$

Let us informally explain the rules for the **if** ... **fi** and **do** ... **od** commands.

The execution of $\textbf{if } gc_1 [\!] \ldots [\!] gc_n \textbf{ fi}$ is performed as follows.
(i) We first choose in a nondeterministic way a guarded command, say $b \rightarrow c$, among $gc_1, \ldots, gc_n$, such that the guard b evaluates to *true*, and then
(ii) we execute the command c.
(iii) If no guarded command among $\{gc_1, \ldots, gc_n\}$ has a guard which evaluates to *true*, the execution of the **if** ... **fi** command is aborted, and the execution of the whole program is aborted.

The execution of $\textbf{do } gc_1 [\!] \ldots [\!] gc_n \textbf{ od}$ is performed as follows.
(i) We first choose in a nondeterministic way a guarded command, say $b \rightarrow c$, among $\{gc_1, \ldots, gc_n\}$ such that the guard b evaluates to *true*, then

(ii) we execute the command c and then go to Step (i).
(iii) If no guarded command among $gc_1, \ldots, gc_n$, has a guard which evaluates to *true*, the execution of the **do** ... **od** command is terminated and the execution continues with that of the command following the **do** ... **od** command.

Note that we gave no rules for the command *abort*. Thus, for every state $\sigma \in State$ and every $\gamma \in State$ no triple $\langle abort, \sigma, \gamma \rangle$ exists in the subset of $Com \times State \times State$ which denotes the operational semantics.

2.2 Operational Semantics of Guarded Commands

The execution of guarded commands is given as a subset of $GCom \times State \times ((Com \times State) \cup \{\mathbf{fail}\})$. A triple in $GCom \times State \times ((Com \times State) \cup \{\mathbf{fail}\})$ is written as $\langle gc, \sigma \rangle \rightarrow \gamma$ and specifies that the guarded command gc from the state σ produces either the new ⟨command, state⟩ pair γ or the value $\gamma = \mathbf{fail}$. The inference rules defining the execution of guarded commands are as follows.

$$(2.1)\ \frac{\langle b,\sigma\rangle \rightarrow true}{\langle b \rightarrow c, \sigma\rangle \rightarrow \langle c,\sigma\rangle} \qquad (2.2)\ \frac{\langle b,\sigma\rangle \rightarrow false}{\langle b \rightarrow c, \sigma\rangle \rightarrow \mathbf{fail}}$$

$$(2.3)\ \frac{\langle gc_1,\sigma\rangle \rightarrow \langle c,\sigma\rangle}{\langle gc_1 \,[\!]\, gc_2, \sigma\rangle \rightarrow \langle c,\sigma\rangle} \qquad (2.4)\ \frac{\langle gc_2,\sigma\rangle \rightarrow \langle c,\sigma\rangle}{\langle gc_1 \,[\!]\, gc_2, \sigma\rangle \rightarrow \langle c,\sigma\rangle}$$

$$(2.5)\ \frac{\langle gc_1, \sigma\rangle \rightarrow \mathbf{fail} \quad \langle gc_2, \sigma\rangle \rightarrow \mathbf{fail}}{\langle gc_1 \,[\!]\, gc_2, \sigma\rangle \rightarrow \mathbf{fail}}$$

Note that when evaluating the guard of a guarded command, the state is *not* changed. This fact is not exploited in the semantic rules given by Winskel [22].

Instead of the above rules (2.3) and (2.4), we can equivalently use the following two rules:

$$(2.3^*)\ \frac{\langle b\rightarrow c,\sigma\rangle \rightarrow \langle c,\sigma\rangle}{\langle b\rightarrow c \,[\!]\, gc_2, \sigma\rangle \rightarrow \langle c,\sigma\rangle} \qquad (2.4^*)\ \frac{\langle b\rightarrow c,\sigma\rangle \rightarrow \langle c,\sigma\rangle}{\langle gc_1 \,[\!]\, b\rightarrow c, \sigma\rangle \rightarrow \langle c,\sigma\rangle}$$

Note also that for all b, c, σ, gc_1, gc_2, we have that:
(i) $\langle b \rightarrow c, \sigma\rangle \rightarrow \langle c, \sigma\rangle$ holds iff $\langle b, \sigma\rangle \rightarrow true$, and
(ii) $\langle gc_1 \,[\!]\, gc_2, \sigma\rangle \rightarrow \langle c, \sigma\rangle$ holds iff there exists $i \in \{1, 2\}$ such that gc_i is $b \rightarrow c$ and $\langle b, \sigma\rangle \rightarrow true$.

Here is Euclid's algorithm for computing the greatest common divisor of M and N using guarded commands (we assume that $M > 0$ and $N > 0$):

$$\{n = N > 0 \wedge m = M > 0\}$$
$$\begin{array}{ll} \mathbf{do} & m \geq n \wedge n > 0 \rightarrow m := m-n \\ [\!] & n \geq m \wedge m > 0 \rightarrow n := n-m \\ \mathbf{od} & \end{array}$$
$$\{gcd(N, M) = \mathit{if}\ n{=}0\ \mathit{then}\ m\ \mathit{else}\ n\}$$

The assertions between curly brackets at the beginning and at the end of the above **do** ... **od** command relate the value of the integer variables before and after the execution of that command.

3 Concurrent Programs: Vectorization

The terminology about concurrent programming is not stable. Some authors consider *concurrent* programs, *distributed* programs, and *parallel* programs to be the same class of programs. Other authors do not.

We assume three forms of concurrency:

(i) concurrency based on *vectorization*,
(ii) concurrency based on *shared variables* (and shared resources), and
(iii) concurrency based on *handshaking communications* (also called *rendez-vous*).

With respect to sequential programs the use of concurrent programs allows us: (i) to get better performance, and (ii) to simulate in a direct way concurrent processes (or agents) which perform computations in real world systems.

3.1 Process Declaration and Process Call

Process declarations and process calls in parallel languages are like procedure declarations and procedure calls in sequential languages, but the *called processes* run together with the *calling processes*.

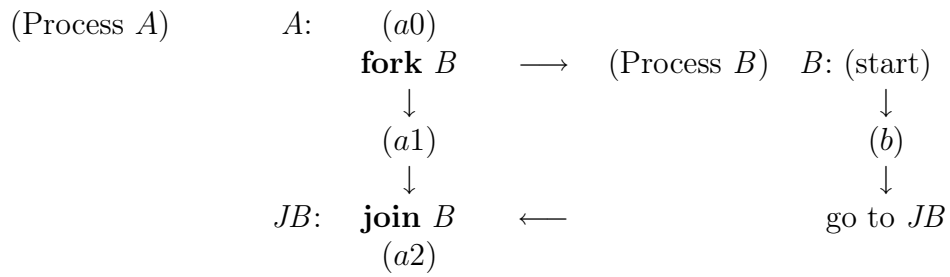
In Concurrent Pascal [5] and Modula [23,24] process declaration can be done at top level only. In this case we say that the processes are *static*.

In Ada [17] a process declaration can be inside an enclosing process declaration. In this case we say that the processes are *dynamic*.

3.2 Concurrency Based on Vectorization Using fork-join

The **fork-join** construct was introduced in [7]. We describe here the following two variants: (1) single **fork-join**, and (2) multiple **fork-join**.

(1) Single **fork-join**:



Process A proceeds after the instruction **join** B, when process B has terminated.

(2) Multiple **fork-join**:

```
(Process A)        A:   (a0)
                        counter := 3;
                        fork B
                          (a1)
                        fork C
                          (a2)
                        go to J
(Process B)        B:   (b)
                        go to J
(Process C)        C:   (c)
                        go to J
                   J:   join counter
                          (a3)
```

Process A proceeds after the instruction **join** *counter*, when both process B and process C have terminated.

Multiple **fork-join** is used, for example, when one has to sum two arrays, say A_1 and A_2, of N elements each, and N is a large number. The sum can be done in parallel by k processes, each summing N/k elements (we assume that N/k is an integer). These k processes are generated by a k-fold **fork**. In particular, process 1 sums the elements of the arrays A_1 and A_2 from position 0 to position $(N/k) - 1$, ..., and process k sums the elements of the arrays A_1 and A_2 from position $N - (N/k)$ to position $N - 1$.

With every **fork-join** program we can associate a multigraph, called *control flow multigraph*, whose nodes are the **fork** and **join** instructions and whose arcs connect nodes according to the usual notion of *control flow* for sequential programs. (We assume that the reader is familiar with this notion.) Note that we consider a multigraph, instead of a graph, because parallel execution may determine two or more arcs between two nodes. For example, the multigraph corresponding to the above program with processes A, B, and C, is depicted in the following Figure 1.

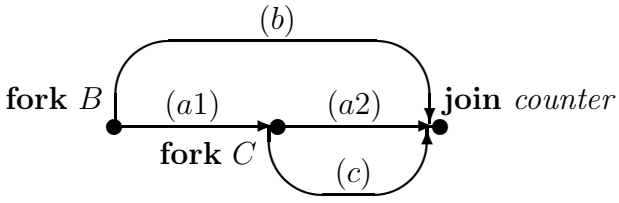


Fig. 1. A control flow multigraph.

The computation continues after a **join** node n iff the computation relative to every arc arriving at n is terminated.

With every control flow multigraph we associate a relation, denoted $\leq$, induced by its arcs. For instance, in the multigraph of Figure 1 we have that: **fork** $C \leq$ **join** *counter*, **fork** $B \leq$ **fork** C, and **fork** $B \leq$ **join** *counter*. If $a_1 \leq \ldots \leq a_n$, for some $n \geq 0$, we say

that a_1 is an ancestor of a_n or, equivalently, a_n is a descendant of a_1. In particular, any node a is an ancestor and a descendant of itself.

3.3 Concurrency Based on Vectorization Using cobegin-coend

The **cobegin-coend** command was introduced in [8]. It has the following syntax:

cobegin S_1; ...; S_n **coend**

Its execution activates n processes, say $P_1, \ldots, P_n$. Process P_1 executes statement S_1, ..., and process P_n executes statement S_n. The **cobegin-coend** command terminates when all processes have terminated.

We assume that every **cobegin** S_1; ...; S_n **coend** command generates a control flow multigraph such as the one depicted in Figure 2.

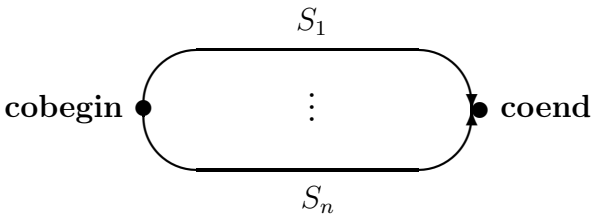


Fig. 2. The control flow multigraph generated by the **cobegin** S_1; ...; S_n **coend** command.

Not all control flow multigraphs that can be generated by **fork-join** constructs (together with sequentialization, **if-then-else**, and **while-do** constructs), can also be generated by **cobegin-coend** commands (together with sequentialization, **if-then-else**, and **while-do** constructs). Obviously, in this generation we consider only the topology of the multigraphs and not the labels of the nodes.

For instance, the multigraph depicted in Figure 3, generated by the program P_{ABC} which consists of the following three processes A, B, and C, *cannot* be generated by **cobegin-coend** commands.

(Process A)	A:	$(a0)$
		fork B
		$(a1)$
		fork C
		$(a2)$
	JB:	**join** B
		$(a3)$
	JC:	**join** C
		go to J
(Process B)	B:	(b)
		go to JB
(Process C)	C:	(c)
		go to JC
	J:	$(a4)$

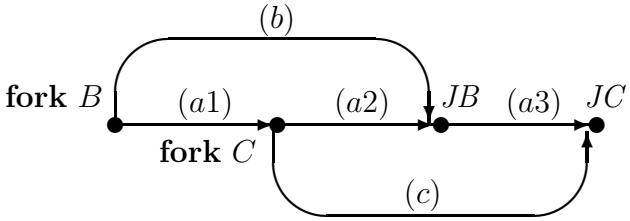


Fig. 3. The control flow multigraph generated by the program P_{ABC}.

Let us consider the following Property Δ of a control flow multigraph:

Property Δ: given any two nodes n_1 and n_2, every descendant of the least common ancestor of n_1 and n_2, is predecessor or a descendant of the greatest common successor of n_1 and n_2.

We have that if a control flow multigraph can be generated by using **cobegin-coend** commands then it satisfies Property Δ. A multigraph which does *not* satisfy Property Δ (and therefore, it cannot be generated by using **cobegin-coend** commands), is the one depicted in Figure 4. Indeed, in Figure 4 node q which is a descendant of the least common ancestor m of the nodes n_1 and n_2, is neither a predecessor nor a descendant of the node p which is the greatest common successor of n_1 and n_2.

Notice that Property Δ is a necessary condition for a control flow multigraph to be generated by using **cobegin-coend** commands, but it is not a sufficient condition, as shown by the multigraph of Figure 3.

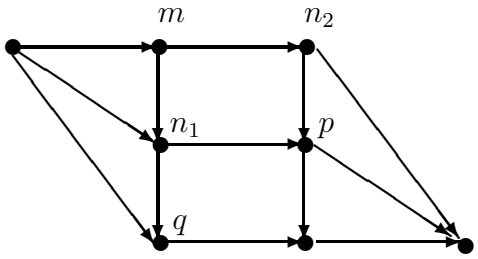


Fig. 4. A control flow multigraph which cannot be generated by using **cobegin-coend** commands.

3.4 Example: Evaluating Recurrence Relations

In order to compute recursive programs which are not linear recursive we can compute in parallel the $m\,(>1)$ recursive calls. For instance, for the Fibonacci numbers we have:

procedure *fibonacci*(int n, int f);
  begin if $n \leq 1$ **then** $f := 1$ **else**
    begin cobegin *fibonacci*$(n-1, f1)$; *fibonacci*$(n-2, f2)$ **coend**;
    $f := f1 + f2$
    end
  end

3.5 Example: Multiplying Matrices

Given the following two $n \times n$ matrices A and B such that

$A = \begin{vmatrix} A_{11}\ A_{12} \\ A_{21}\ A_{22} \end{vmatrix}$ and $B = \begin{vmatrix} B_{11}\ B_{12} \\ B_{21}\ B_{22} \end{vmatrix}$, we have that $C = \begin{vmatrix} C_{11}\ C_{12} \\ C_{21}\ C_{22} \end{vmatrix}$ is the product $A \times B$ iff

$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21},\ \ C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22},$
$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21},\ \ C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}.$

Thus, we can compute C by performing in parallel 8 multiplications of $n/2 \times n/2$ matrices as follows. For reasons of simplicity we assume that n is a power of 2.

procedure *mult*(int n, int $A[1..n, 1..n]$, int $B[1..n, 1..n]$, int $C[1..n, 1..n]$);
  begin if $n = 1$ **then** $C := A \times B$ **else**
    begin
    cobegin
      mult$(n/2,\ A_{11},\ B_{11},\ C1)$; *mult*$(n/2,\ A_{12},\ B_{21},\ C2)$;
      mult$(n/2,\ A_{11},\ B_{12},\ C3)$; *mult*$(n/2,\ A_{12},\ B_{22},\ C4)$;
      mult$(n/2,\ A_{21},\ B_{11},\ C5)$; *mult*$(n/2,\ A_{22},\ B_{21},\ C6)$;
      mult$(n/2,\ A_{21},\ B_{12},\ C7)$; *mult*$(n/2,\ A_{22},\ B_{22},\ C8)$
    coend;
    cobegin
      $C11 := C1 + C2$; $C12 := C3 + C4$;
      $C21 := C5 + C6$; $C22 := C7 + C8$
    coend;
    $C :=$ *arrange*$(C11, C12, C21, C22)$
    end
  end

where *arrange* is a function that given four $n/2 \times n/2$ matrices, say $C11$, $C12$, $C21$, and $C22$, constructs the $n \times n$ matrix $C = \begin{vmatrix} C11\ C12 \\ C21\ C22 \end{vmatrix}$.

4 Concurrent Programs Based on Shared Variables

4.1 Preliminary Example: Prefix Sums

Let us present a preliminary example of a concurrent program based on shared variables. We are given a sequence $\langle x_0, x_1, \ldots, x_{n-1}\rangle$ of n numbers and we want to compute the sequence $\langle s_0, s_1, \ldots, s_{n-1}\rangle$ also of n numbers such that:

$s_0 = x_0$,
$s_1 = x_0 + x_1$,
$s_2 = x_0 + x_1 + x_2$, ..., and
$s_{n-1} = x_0 + x_1 + \ldots + x_{n-1}$.

This computation can be performed by the following program, where we assume n processors $P_0, P_1, \ldots, P_{n-1}$. For $i = 0, 1, \ldots, n-1$, processor P_i initially holds x_i and, finally, P_i holds s_i. To make subscripts more readable we write $2\hat{}t$, instead of 2^t.

par-for $i = 0$ **to** $n-1$ **do** $s_i := x_i$ **od**;
for $t = 0$ **to** $(\log n) - 1$ **do**
  par-for $k = 2\hat{}t$ **to** $n-1$ **do** $s_k := s_k + s_{k-2\hat{}t}$ **od** (†)
od

In this program the construct

par-for $i = 0$ **to** p **do** $body_i$ **od**

is executed by making every processor P_i, for $i = 0, \ldots, p$, to execute $body_i$ independently and in parallel. The statement terminates when all processors have terminated. Thus, in line (†) above the values of s_k and $s_{k-2\hat{}t}$ in the expression $s_k + s_{k-2\hat{}t}$ are the results of the previous iteration of the **par-for** loop of the same line (†). In general:

par-for $i = 1$ **to** p **do** $body_i$ **od**

is assumed to be equivalent to:

cobegin $body_1$; ...; $body_p$ **coend**.

The above program uses the so called *recursive doubling* technique. This name comes from the fact that the number of processors which have no sums to perform, doubles at each execution of the body of the outermost **par-for** loop. The flow of data, when $n = 8$, is depicted as in Figure 5. There are 8 processors, one for each column. Each row shows the value computed by the processors at the corresponding time. Values arriving at a processor along incoming arcs are added together, and copies of the results are sent along the outgoing arcs.

The complexity or *cost* of the above program measured in time × number of processors is $O(\log n) \times O(n)$, i.e., $O(n \log n)$. This cost is *not* optimal because if we use one processor only, the time × number of processors complexity is $O(n) \times O(1)$, that is, $O(n)$.

Notice that the above program which is used for summing up the numbers of a sequence, can also be used for computing the sequence $\langle p_0, p_1, \ldots, p_{n-1}\rangle$ of products such that: $p_0 = x_0$, $p_1 = x_0 \times x_1$, ..., and $p_{n-1} = x_0 \times x_1 \times \ldots \times x_{n-1}$, by replacing $s_k := s_k + s_{k-2\hat{}t}$ by$s_k := s_k \times s_{k-2\hat{}t}$. Indeed, that program can be used for any binary operation which is associative.

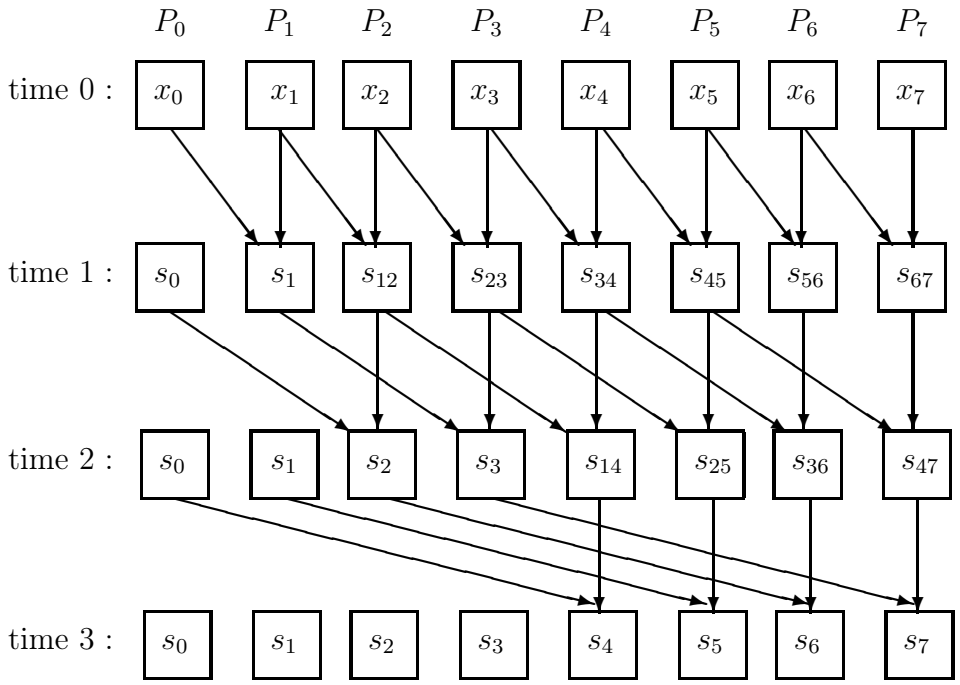


Fig. 5. The computation of prefix sums of the 8 numbers: x_0, x_1, ..., x_7. For $0 \leq i \leq 7$, s_i denotes $x_0 + x_1 + \ldots + x_i$. $s_0 = x_0$. For $0 \leq i \leq j \leq 7$, s_{ij} denotes $x_i + x_{i+1} + \ldots + x_j$.

Finally, let us notice that sometimes the distinction between concurrent computations which are based on vectorization and those which are based on shared memory is not so sharp. Indeed, this distinction very much depends on the amount of interactions among the various processes (or processors) during the execution of the bodies of the **par-for** loops involved in the computation. If there is little interaction, we prefer to say that the computation is based on vectorization, otherwise, we say that it is based on shared memory.

4.2 Prefix Sums Revisited

Let us consider again the example of the previous Section 4.1. We will present two new programs for the computation of the prefix sums of a given sequence $x_0, \ldots, x_{n-1}$ of n numbers. Both programs run on a tree of processors (see also Figure 6 where we consider the case for $n = 8$), but they have different complexity measured in terms of time $\times$ number of processors. The first program has $O(n \log n)$ complexity (which is not optimal), while the second program has $O(n)$ complexity (which is optimal). The optimal complexity is $O(n)$ because every program should look at every number of the sequence and the length of the sequence is n.

Let us consider the first program $P1$. We assume that we have n processors $P_0, \ldots$, P_{n-1}, each one at a leaf of a binary tree of processors, and we also have $n-1$ processors, each one at a non-leaf node of that binary tree. For reasons of simplicity, we assume that

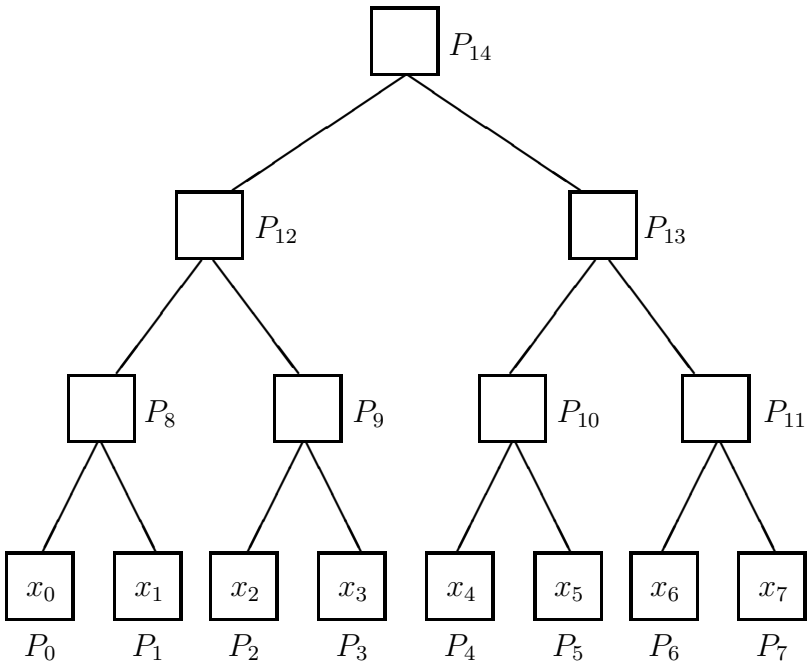


Fig. 6. The computation of prefix sums of a sequence $x_0, \ldots, x_7$ of $n\,(=8)$ numbers on a tree of $2n-1\,(=15)$ processors: $P_0, \ldots, P_{14}$.

n is a power of 2 (see Figure 6 where $n=8$). The parallel computation proceeds according to the following rules (see also Figure 7) in an asynchronous way, that is, a rule is applied in a node of the tree whenever it can be applied, regardless at what happens in other nodes:

- rule for the root:

*Rule R*1.*down*: the root, after receiving a value v from its left child, sends v to the right child;

- rules for the internal nodes:

*Rule R*2.*up*: an internal node, after receiving the values v_1 and v_2 from its children, sends their sum v_1+v_2 to the father and sends the value v_1 from the left child to the right child;

*Rule R*2.*down*: an internal node, after receiving a value v from its father, sends v to the two children;

- rules for the leaves:

*Rule R*3.*up*: a leaf sends its value x to its father, and this operation is performed only once at the beginning of the computation;

*Rule R*3.*down*: a leaf, after receiving a value v from its father, sums v to its own value.

Notice that throughout the computation: (i) one value only is received by the root and this value is the only value sent by the root down to its children, (ii) one value only is sent up by any non-root node to its father, and (iii) one value only is sent down by any non-leaf node to its children.

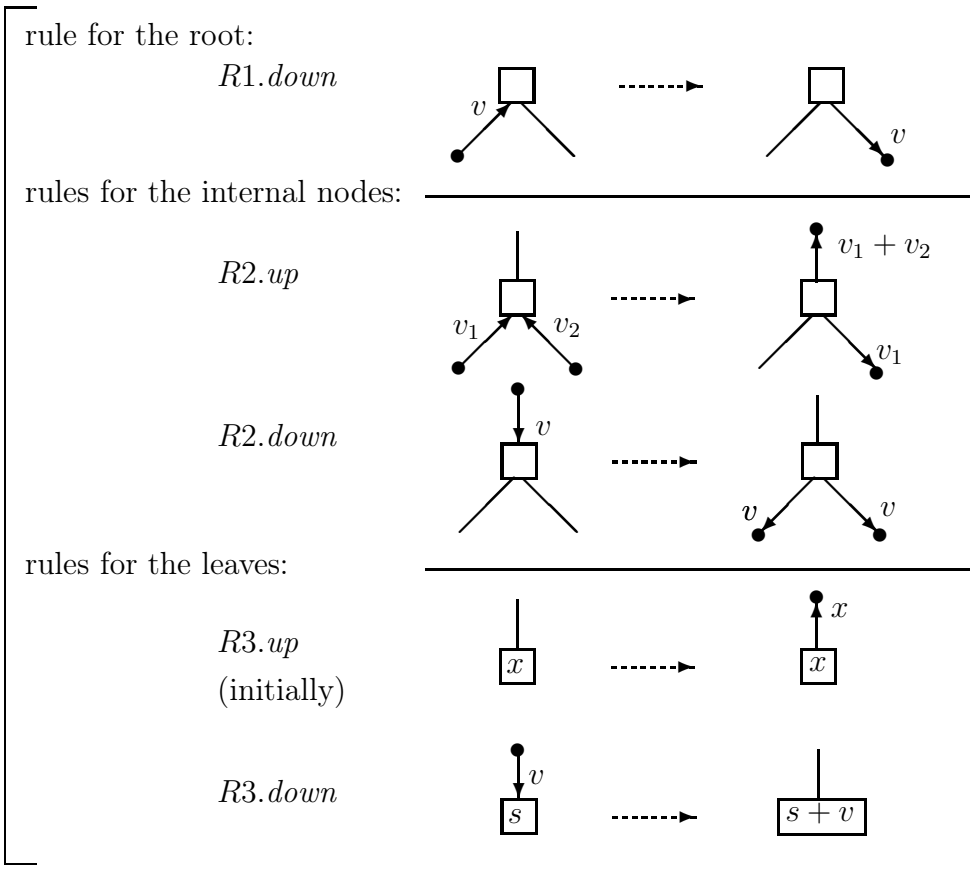


Fig. 7. The rules for the computation of prefix sums for Program $P1$. The initial value of s is x. The rule for the root is an instance of the rules for the internal nodes, because the root has no father.

The time taken by this program is $O(\log n)$ because a value has to travel up to the root and then down to the leaves. The number of processors is $2n-1$. Thus, the time $\times$ number of processors complexity is $O(n \log n)$.

Now we present a second program $P2$ for the computation of the prefix sums. This program $P2$ is like program $P1$, except that the rules for the computation at the leaves are modified.

We assume that we are given a sequence of n numbers, and we have N processors, named $P_0, \ldots . P_{N-1}$, placed at the N leaves of the binary tree of processors from left to right. We have some more $N-1$ processors located at the non-leaf nodes of the binary tree. (For reasons of simplicity, we assume that n is a power of 2.) Thus, the total number of processors is $2N-1$. Without loss of generality, we also assume that n/N is an integer. Let q be that integer. We divide the given sequence $x_0, \ldots, x_{n-1}$ in N subsequences of q numbers each, and we allocate each of these subsequences in left-to-right order to each of the N processors at the leaves of the binary tree. Each leaf has an associated accumulator initialized to 0.

Here are the new rules for the computation performed at the leaves (see Figure 8):

$\overline{R3}.up$: a leaf computes the prefix sums of the associated subsequence of q numbers and sends the sum of all these q numbers of that subsequence to its father;

$\overline{R3}.down$: a leaf, after receiving a value from its father, sums it to its accumulator r;

$\overline{R3}.end$: the leaf with processor P_i, after receiving $b(i)$ values from its father, sums the final value of r to each value of the prefix sums computed at Step $\overline{R3}.up$. $b(i)$ is the number of right-child arcs to take for going from the root to the leaf. It can be computed as follows:

$b(0) = 0$
$b(2n) = b(n)$
$b(2n+1) = b(n) + 1$

It is the case that $b(i)$ is equal to the number of 1's in the binary expansion of i. The *total* number of values received from the leaf with processor P_i is $b(i)$, thus, the rule $\overline{R3}.end$ is applied in each leaf at the end of the computation (hence, the name of the rule).

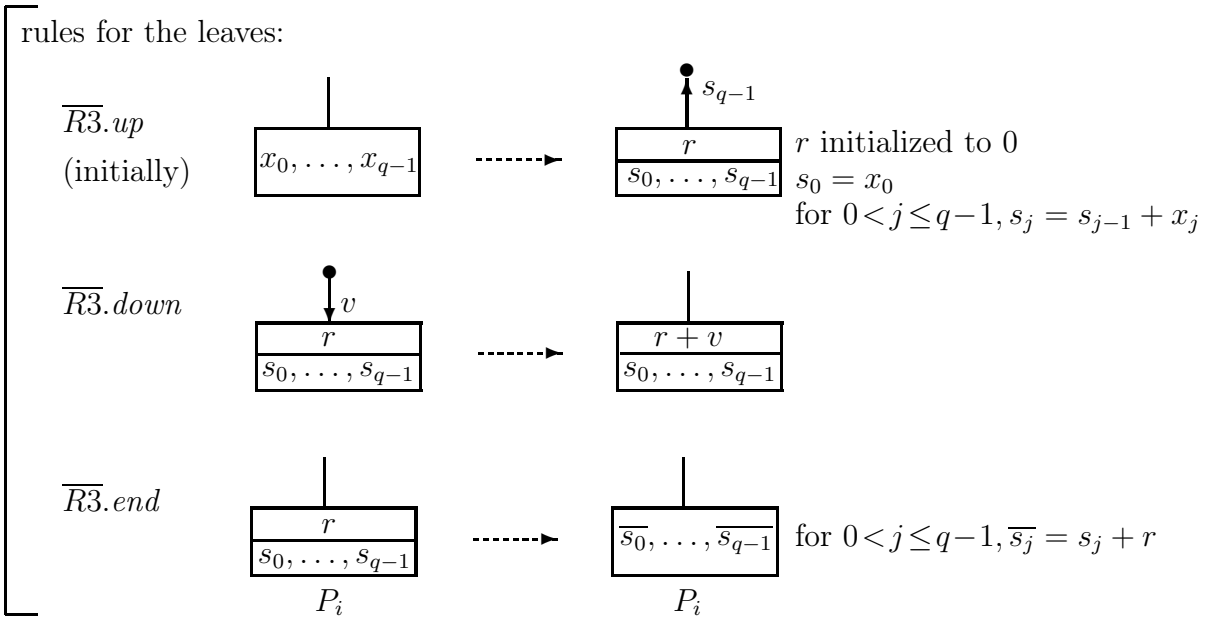


Fig. 8. The rules for the leaf computation of prefix sums for Program $P2$. Rule $\overline{R3}.end$ is applied when $b(i)$ values are arrived at the leaf with processor P_i. The function $b(i)$ is defined as follows: $b(0) = 0$, $b(2n) = b(n)$, $b(2n+1) = b(n) + 1$.

We have that: (i) $\overline{R3}.up$ requires $O(n/N)$ time, (ii) rules $R1.down$, $R2.up$, and $R2.down$ require $O(\log N)$ time as for program $P1$, and (iii) rules $\overline{R3}.down$ and $\overline{R3}.end$ require $O(n/N)$ time. Thus, the total time is $O(n/N) + O(\log N)$. The total number of processors is $2N-1$. The time $\times$ number of processors complexity of Program $P2$ is: $O(n) + O(N \log N) = O(n + N \log N)$, because $O(f(n) + g(n)) = O(f(n)) + O(g(n))$. Now $O(n + N \log N)$ is equal to $O(n)$, if we take $N = n/\log n$. Since the base of the logarithm is not significant, we may take it to be 2. This means that if the total length n of

the sequence is 1000 then N may be taken to be about $1000/\log_2 1000 \approx 100$ to ensure that the total running time of the algorithm is linear.

Notice that the parallel algorithm for computing the prefix sums is not *fully parallel*, in the sense that some operations are done sequentially. For instance, the computation of the prefix sums of the subsequences in each of the leaves of the tree of processors is done sequentially, but it can also be done in parallel if enough processors are available.

Exercise 1. We leave to the reader to study: (i) the prefix sums computation on a mesh architecture, (ii) the sorting problem in parallel, and (iii) the Owicki-Gries calculus for proving correctness of parallel programs with shared variables.

4.3 Semaphores

Semaphores are used for synchronizing the activities of several processes which run concurrently. They were introduced by Prof. E. W. Dijkstra in [8]. A semaphore s is an integer variable which can be manipulated by the following two operations only:
(i) the *wait*(s) operation (also called $P(s)$ operation), and
(ii) the *signal*(s) operation (also called $V(s)$ operation).

The initialization of a semaphore to an integer value, say N, is done by the clause **initial**(N) added to its definition (see below).

Note 2. Prof. E. W. Dijkstra was Dutch and in Dutch 'to pass' is 'passeren' (thus, the letter P) and 'to release' is 'vrygeven' (thus, the letter V). □

---

var s : *semaphore* **initial**(N); /* initialization. N is an integer */
wait(s) : **while** $s \leq 0$ **do** *skip* **od**; $s := s-1$;
signal(s) : $s := s+1$;

In the definition of the *wait*(s) operation we have used the construct '**while** b **do** c **od**', instead of the construct '**while** b **do** c', for denoting in an unambiguous way the body of the while-loop. The *wait*(s) operation is equivalent to:

α: **if** $s \leq 0$ **then** *goto* α **else** $s := s-1$

In order to understand the behaviour of semaphores it is important to know which operations are atomic and which are not. Atomic operations are defined as follows.

Definition 1. An operation (or statement or sequence of statements) performed by a process is said to be *atomic* iff during its execution no other operation can be executed by any other process. We also say that an atomic operation (or statement or sequence of statements) is performed in *mutual exclusion* with respect to every other operation.

Atomicity of operations (or statements or sequences of statements) is enforced by the hardware. The *wait*(s) operation is *not* atomic, but each execution of the statement

if $s \leq 0$ **then** *goto* α **else** $s := s-1$

is atomic. Thus, in particular, when a process finds that s has positive value, it performs the assignment $s := s-1$ on that value of s, because no other process can intervene between the test and the assignment.

We say that the *wait*(s) operation is *completed* when the assignment $s := s-1$ is performed.

Between two consecutive executions of the statement

if $s \leq 0$ **then** *goto* α **else** $s := s-1$

performed by a process, say P, (and during the first of the two executions we have that $s \leq 0$), a different process, say Q, may perform an operation, and if this operation is a *signal*(s) operation then P may complete its *wait*(s) operation because it may find that s is positive and performs the assignment $s := s-1$.

The operation *signal*(s) is *atomic*, that is, no other operation occurs on the variable s while the *signal*(s) operation takes place.

A semaphore is said to be *binary* iff the value of s belongs to $\{0, 1\}$. For any binary semaphore s, when s is 1, subsequent *signal*(s) operations do not have any effect. A binary semaphore may be realized by a boolean variable s which, after its initialization to a boolean value $B \in \{false, true\}$, can be manipulated only by the two operations *wait*(s) and *signal*(s) defined as follows.

---

var s : *semaphore* **initial**(B); /* initialization. $B \in \{true, false\}$ */
wait(s) : **while** $s = false$ **do** *skip* **od**; $s := false$;
signal(s) : $s := true$;

A semaphore which is not binary, is also called a *counting semaphore*.

The use of semaphores may determine the so called *busy waiting* phenomenon, which we now illustrate in the case of counting semaphores. Busy waiting occurs when a process while performing the *wait*(s) operation keeps on testing whether or not $s \leq 0$, even if at a previous instant in time, it found s to be non-positive and since then its value has not been changed.

In order to avoid busy waiting we do as follows. We suspend every process which, while performing a *wait*(s) operation, finds s to be non-positive, and we tell every process which performs a *signal*(s) operation to wake up a suspended process. Thus, the implementation of a semaphore s includes an associated (possibly empty) set of waiting processes, that is, processes which are executing a *wait*(s) operation on the semaphore s and have not yet completed that operation. Every process in that set is said to be *suspended on the semaphore s*.

When a *signal*(s) operation is executed on the semaphore s, then *exactly one* process in the set associated with s is resumed. In order to satisfy a fairness requirement, that set of waiting processes is usually served using the first-in-first-out policy, that is, it is structured as a *queue*.

Obviously, some properties of the concurrent programs which use a semaphore, may depend on the policy of serving the set of processes associated with the semaphore. One such property is, for instance, the freedom of *starvation* (see end of Section 4.4). Notice, however, that when a semaphore is used for ensuring mutually exclusive access to a shared resource (see Section 4.4), that property should *not* depend on the policy of serving the set of processes associated with the semaphore.

4.4 Semaphores and Test-and-Set Operations for Mutual Exclusion

Let us consider a set of n processes which run concurrently. For $i = 1, \ldots, n$, the i-th process executes a program P_i that has a section, called *critical section*, which should be executed in a mutually exclusive way. This means that the entire sequence of instructions of a critical section should be executed by a process, while the control point of every other process in the given set of processes is at an instruction outside a critical section.

When mutual exclusion is ensured, we say that at most one process at any time is *inside* a critical section. The following program realizes the mutual exclusion among n processes by using the binary semaphore *mutex*.

---

var *mutex* : *semaphore* **initial**(1); /* *mutex* $\in \{0, 1\}$ */

process $P_1(...)$
begin ...; *wait*(*mutex*); critical section 1; *signal*(*mutex*); ... **end**;
...

process $P_n(...)$
begin ...; *wait*(*mutex*); critical section n; *signal*(*mutex*); ... **end**;

The semaphore *mutex* is like a token which is put in a basket or is taken from it. There is one basket and one token only. At any instant in time, in the basket there is at most one token, and initially, the token is in the basket (see **initial**(1)). The fact that two or more processes cannot be inside their critical sections at the same time, is a consequence of how semaphores behave (see Section 4.3). Thus, mutual exclusion may be ensured by the use of semaphores.

If the set of processes waiting on the semaphore *mutex* is served using the first-in-first-out policy, then there is no starvation.

Mutual exclusion may also be ensured by the use of *test-and-set* instructions as we now illustrate. The instruction *test-and-set*(m, l) acting on the *global* variable m (that is, a variable which can be read or written by every process) and the local variable l (that is, a variable which can be read or written only by the process which has defined it), is a pair of assignments of the form:

$l := m$; /* copying the old value of m into a *local* variable l */
$m := 0$ /* setting the new value of m to 0 */

with the condition, imposed by the hardware, that no other instruction may be executed on the variables m and l in between the two assignments of the pair.

Now we present a program which ensures mutual exclusion among n processes using *test-and-set* operations and the global variable m. If $m = 0$ then a process is inside its critical section, and if $m = 1$ then no process is inside a critical section.

```
var m : 0..1 initial(1);        /* m ∈ {0,1} */
  process P_1(...)
    begin var l_1 : 0..1;       /* local variable l_1 for copying the global variable m */
          ...;
          α_1 : test-and-set(m, l_1); if l_1=0 then goto α_1;
          critical section 1;
          m := 1; ...
    end;
  ...
  process P_n(...)
    begin var l_n : 0..1;       /* local variable l_n for copying the global variable m */
          ...;
          α_n : test-and-set(m, l_n); if l_n=0 then goto α_n;
          critical section n;
          m := 1; ...
    end;
```

In this program mutual exclusion is ensured because when $m = 0$, we have that a process is inside its critical section and no other process may enter. The variable m is initialized to 1 by the clause **initial**(1)**.** However, in this program there is the possibility of starvation, that is, a process which wants to enter its critical section, can never do so it because it is always overtaken by some other process.

4.5 Semaphores for Mutual Exclusion for Producers and Consumers

Let us consider a set of processes. A process is said to be a *producer* if it calls the procedure *send*, and it is said to be a *consumer* if it calls the procedure *receive* (see below).

Let us consider a *circular buffer* of N cells, from cell 0 to cell N–1 which is implemented as an array, called *buffer*, of N elements (see Figure 9). We assume that a cell of the buffer can be occupied by a *message*. There are two indexes: *in* and *out*, and these indexes are used for inserting and extracting messages to and from the buffer, respectively.

Initially the N cells of the buffer are all free, i.e., no message is in the buffer. For instance, a message may be a text page which is generated by a process (viewed as a producer) and is printed by another process (viewed as a consumer).

The use of the circular buffer by some producer and consumer processes is regulated by the following procedures and shared variables.

Variables shared among all processes (as usual, the clause **initial**(n) in the definition of a variable initializes that variable to the value n):

```
var buffer : array [0..N−1] of message;      /* the circular buffer */
    in :  0..N−1  initial(0);
    out : 0..N−1  initial(0);
```

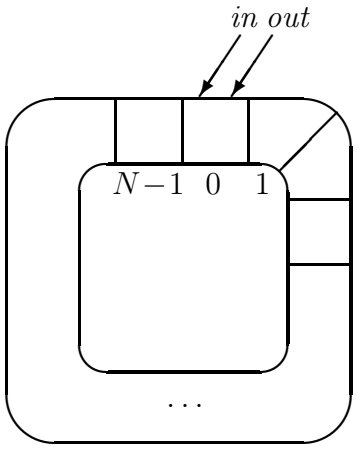


Fig. 9. A circular buffer of size N.

```
mutex_s : semaphore  initial(1);        /* mutex_s ∈ {0,1} */
mutex_r : semaphore  initial(1);        /* mutex_r ∈ {0,1} */
free_cells : semaphore  initial(N);     /* free_cells ∈ {0,...,N} */
messages_in : semaphore  initial(0);    /* messages_in ∈ {0,...,N} */
```

Procedure called by a *producer*:

```
procedure send(x : message);
  begin wait(free_cells); wait(mutex_s);
        buffer[in] := x;   in := (in+1) mod N;        /* critical section 1 */
        signal(mutex_s); signal(messages_in);
  end
```

Procedure called by a *consumer*:

```
procedure receive(var x : message);
  begin wait(messages_in); wait(mutex_r);
        x := buffer[out];   out := (out+1) mod N;     /* critical section 2 */
        signal(mutex_r); signal(free_cells);
  end
```

The semaphore *mutex_s* forces that *at most one* process at a time executes the *send* procedure. Analogously, the semaphore *mutex_r* forces that *at most one* process at a time executes the *receive* procedure.

Remark 1. Notice that it is not enough to consider the value of the variables *in* and *out* to ensure a correct behaviour of the circular buffer. Indeed, if $in \neq out$ then a producer and a consumer will act in different cells without interference. However, since in general, there is more than one producer and more than one consumer, we have to ensure their mutual exclusive access to the circular buffer by using semaphores. □

Having the semaphore *free_cells* is like having a basket and N tokens, called *free_cells* tokens, which initially are all in the basket (see **initial**(N)). After the execution of *wait*(*free_cells*), a *free_cells* token is taken away from the basket (which means that there is one less free cell in the circular buffer), and after the execution of *signal*(*free_cells*), a *free_cells* token is put back in the basket (which means that there is one more free cell in the circular buffer).

Having the semaphore *messages_in* is like having a basket and N tokens, called *messages_in* tokens, each of which initially is not in the basket (see **initial**(0)). After the execution of *signal*(*messages_in*) by the *send* procedure, a *messages_in* token is placed in the basket (which means that at least one message is in the circular buffer), and after the execution of *wait*(*messages_in*) by the *receive* procedure, a *messages_in* token is taken away from the basket (which means that one message has been extracted from the circular buffer).

These are four possible situations:

(i) either no process is in a critical section, or

(ii) one producer is in its critical section, or

(iii) one consumer is in its critical section, or

(iv) one producer is in its critical section and one consumer is in its critical section.

In order to avoid situation (iv) it is enough to replace the two semaphores *mutex_s* and *mutex_r* by a single semaphore, say *mutex*.

Since *free_cells* is initially N, at any time we have that:

number of messages inserted during the whole history by the producers
$\leq N+$ number of messages extracted during the whole history by the consumers.

4.6 Private Semaphores for Establishing a Policy to Resume Processes

Many processes may be in a busy waiting status when testing the value of the semaphores. In order to avoid busy waiting, we may suspend processes. Then, we have to establish a policy for resuming a process among the ones which are suspended. Using a suitable policy we may avoid starvation as well. To this aim we may use *private semaphores*, as indicated in the example below.

Definition 2. A semaphore s is said to be *private to a process* iff only that process can execute the *wait*(s) operation.

Let us assume that there are K processes (from 1 to K), and a shelf with capacity N, modeled by an array, called *shelf*, of N *cells* (from 0 to $N-1$), depicted in the following Figure 10.

Any process, say P_i, if it is a producer, requires m cells (not necessarily contiguous) from the shelf to place m cakes. We assume that $0 < m \leq N$. This action is performed by P_i by calling the procedure *put*(m : *integer*, i : $1..K$) (see below). If the request of m cells cannot be granted (maybe because m is greater than the current available number of cells), it is recorded by the assignment *request*$[i] := m$, that is, by making the i-th element of a given array, called *request*, equal to m. A consumer process which wants to

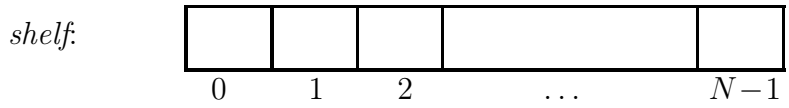


Fig. 10. The array *shelf* with N cells.

get m cakes from the shelf, calls the procedure $get(m : integer)$ (see below). We assume that $0 < m \leq N$.

The value of the variable *free_cells* is the number of cells of the shelf which do not have a cake (these cells are also called *free cells*).

Beside the rule which states that the $put(m, i)$ and $get(m)$ procedures should be executed in a mutually exclusive way, we use the following rules for regulating the activity of the producers and the consumers.

Rule R1. A producer which wants to execute $put(m, i)$, that is, wants to place m cakes on the shelf, can do so if no producer is suspended and m is not greater than the number of free cells.

Rule R2. A producer which wants to execute $put(m, i)$, is suspended if there are other producers which are suspended or m is greater than the number of free cells at that time.

Rule R3. If a producer P_i which wants to execute $put(m, i)$, is suspended, it leaves its request which was not granted by setting $request[i] := m$. Consumers are never suspended.

Rule R4. During the execution of the procedure $get(m)$, the consumer in its critical section uses the following policy for activating suspended producers:

α : the consumer considers among the suspended producers the one, say P_k,
with the largest request, say $request[k]$, and
if $request[k]$ cannot be granted
(that is, the number of free cells is smaller than $request[k]$)
then P_k is left suspended, no other producer is activated, and
the consumer exits its critical section,
else P_k is activated, P_k puts as many as $request[k]$ cakes on the shelf, and
the consumer goes back to α and stays in its critical section. □

These rules ensure neither absence of deadlock nor absence of starvation.

The variables shared among all processes are the following ones:

```
var shelf : array [0..N−1] of cell;                          /* the shelf */
    request : array [1..K] of integer initial_all(0);        /* array [1..K] of integers */
    free_cells : integer  initial(N);                        /* free_cells ∈ {0, ..., N} */
    mutex : semaphore  initial(1);                           /* mutex ∈ {0, 1} */
    done :  semaphore  initial(0);                           /* done ∈ {0, 1} */
    priv_sem : array [1..K] of semaphore initial_all(0);     /* array [1..K] of 0..1 */
```

A *producer* process P_i which wants to put $m\,(>0)$ cakes on the shelf, executes the following two instructions:

put(*m*,*i*); /* if *free_cells* $< m$ then no cakes are put on the shelf */
if *request*[*i*] $= m$ **then begin** P_i puts m cakes in the shelf; *signal*(*done*) **end**;

where: (i) no jump from other instructions to the **if-then** statement is allowed, and (ii) the procedure *put*(m, i) is as follows:

```
procedure put(m : integer, i : 1..K);
  begin wait(mutex);
        if ∀i, 1≤i≤K, request[i] = 0 and free_cells ≥ m
            then begin free_cells := free_cells−m;
                       P_i puts m cakes on the shelf;
                       signal(priv_sem[i]);
                 end
            else request[i] := m;
        signal(mutex);
        wait(priv_sem[i]);
  end
```

A *consumer* process P_j which wants to get $m\,(>0)$ cakes from the shelf, executes the following instruction:

get(m);

where the procedure *get*(m) is as follows:

```
procedure get(m : integer);
  begin wait(mutex);
        P_j gets m cakes from the shelf;          /* ——— (1) */
        free_cells := free_cells + m;
        while ∃i, 1≤i≤K, request[i] ≠ 0 do
              choose a suspended producer with largest request, say P_k;
              if request[k] ≤ free_cells
                  then begin signal(priv_sem[k]);
                             wait(done);
                             free_cells := free_cells − request[k];
                             request[k] := 0;
                       end
                  else goto η;  od;
     η : signal(mutex);
  end
```

Remark 1. If all processes are producers there is deadlock, in the sense that we get into a situation in which no process can proceed, although each of them wants to proceed.

Indeed, if all processes are producers, at a given moment the buffer gets filled with cakes and subsequent producers will be suspended forever. □

Remark 2. $\forall i,\ 1 \leq i \leq K$, *request*$[i] = 0$ means that there is no suspended producer. □

Remark 3. In order to avoid the possibility that a consumer executes the procedure $get(m)$ with m which is greater than the number of cakes available on the shelf, we should replace line (1) of that procedure, that is:

P_j gets m cakes from the shelf;

by:

(a) **if** *free_cells* $+ m > N$ **then** *goto* η **else** P_j gets m cakes from the shelf;

or else:

(b) **if** *free_cells* $+ m > N$ **then** $m := N -$ *free_cells*;
P_j gets m cakes from the shelf;

In Case (a) the consumer is forced to exit the critical section without getting any cake, while in Case (b) the consumer is allowed to get all the cakes which are on the shelf (notice that during the execution of statement at line (1), only assignments to the array *shelf* are performed). □

Remark 4. Let us assume that a producer, say P_i, is suspended outside its critical section because it executes *signal*(*mutex*) and then *wait*(*priv_sem*$[i]$) and we have that *request*$[i] = m$. This means that during the execution of the *put* procedure, the **else** branch was taken. When a consumer, say P_j, has executed *signal*(*priv_sem*$[i]$), then P_i executes:

if *request*$[i] = m$ **then begin** P_i puts m cakes on the shelf; *signal*(*done*) **end**

where *request*$[i] = m$ holds. Thus, P_i puts m cakes on the shelf and then it executes *signal*(*done*). This last *signal* operation allows the consumer P_j, still in its critical section, to continue running and thus, it may update the value of *free_cells* and reset the value of *request*$[i]$ to 0 (because now process P_i has no longer a pending request).

Notice also that while the consumer P_j is waiting for the *signal*(*done*) to be executed by the producer P_i, no other producer or consumer may enter a critical section, because P_j is waiting in its critical section and no *signal* operation has been done on the semaphore *mutex*. □

Remark 5. If $\forall i,\ 1 \leq i \leq K$, *request*$[i] = 0$ **and** *free_cells* $\geq m$ holds, then a producer, say P_i, puts m cakes on the shelf, performs *signal*(*priv_sem*$[i]$), and exits its critical section. Then, it also exits the *put* procedure, because the *signal*(*priv_sem*$[i]$) operation allows the completion of the *wait*(*priv_sem*$[i]$) operation. Then when process P_i executes:

if *request*$[i] = m$ **then begin** P_i puts m cakes on the shelf; *signal*(*done*) **end**

we have that *request*$[i] \neq m$ and, P_i correctly neither puts cakes on the shelf nor executes *signal*(*done*). □

Remark 6. Let us assume the following definitions for the producer process and the consumer process. They are variants of the definitions we have given above and do *not* use the semaphore *done*.

A *producer* process P_i which wants to put m (> 0) cakes on the shelf, executes the following two instructions:

put(m,i); /* if *free_cells* $< m$ then no cakes are put on the shelf */
if *request*[i] $= m$ **then** P_i puts m cakes in the shelf; /* ——— (2) */

The procedure *get*(m) for a *consumer* process is:

```
procedure get(m : integer);
  begin wait(mutex);
        P_j gets m cakes from the shelf;
        free_cells := free_cells + m;
        while ∃i, 1≤i≤K, request[i] ≠ 0 do
              choose a suspended producer with largest request, say P_k;
              if request[k] ≤ free_cells
                 then begin signal(priv_sem[k]);        /* ——— (3) */
                            free_cells := free_cells − request[k];
                            request[k] := 0;
                      end
                 else goto η;  od;
     η : signal(mutex);
  end
```

These variant definitions for a producer process and the procedure *get*, are *not* correct, because the following undesirable situations (A) or (B) may arise.
(A) After the execution of the statement (3), the consumer may execute the subsequent two statements (which update the values of *free_cells* and *request*[k]) before the producer process which has been activated may execute the statement (2). The activated producer will not put its cakes on the shelf because we have that *request*[i] $\neq m$ (indeed, m is larger than 0 and the consumer sets *request*[i] to 0).
(B) Assume that when a consumer process, say P_j, is in its critical section, it activates a producer process P_k which is the only one which is suspended. Then process P_j exits its critical section because no more suspended producers are present. Before P_k executes the statement (2), another producer process, say P_z, enters its critical section finding the new values of *free_cells* and *request*[k] as they should be after P_k would have put its cakes on the shelf. This allows P_z to put cakes in the cells which should be used by P_k. □

4.7 Critical Regions and Conditional Critical Regions

The above section shows that the proofs of correctness of concurrent programs that use semaphores can be rather difficult. To allow easier proofs of correctness, some new language constructs have been proposed such as the *critical regions* (also called *regions*), the *conditional critical regions* (also called *conditional region*), and the *monitors*.

In this section we will consider the critical region construct and the conditional critical region construct.

Critical Regions

The critical region construct was introduced in [4]. It looks as follows:

---

var v : **shared** T; /* v is a variable of type T and
can be shared among several processes */
region v **do** S **end**

This construct ensures that during the execution of the statement S, called *critical region*, no other process may access the variable v. We also assume that:

(i) for each variable v, at most one process at a time can execute a statement of the form **region** v **do** S **end**,

(ii) if a process wants to enter a critical region S, that is, it wants to execute the statement **region** v **do** S **end**, and no other process has access to the variable v, then it will be allowed to do so within a finite time, and

(iii) if S does not contain any critical region construct, then the execution time of the construct **region** v **do** S **end** is finite.

Notice that the variable v need not occur inside S.

If a process executes **region** v **do** S_1 **end** at about the same time another process executes **region** v **do** S_2 **end**, then the result is equivalent to either $S_1; S_2$ or $S_2; S_1$, depending on the first process which enters its critical region according to the actual scheduling performed by the system.

The system associates a queue, say Qv, with each shared variable v, so that if a process, say P, must be suspended because another one is inside a critical region associated with v, then P is inserted in the queue Qv, and it will be resumed according to the first-in-first-out policy.

By using critical regions we can get mutual exclusion for the access to critical sections as follows:

---

var v : **shared** T;
process $P_1(...)$;
  begin ...; **region** v **do** critical section 1; **end**; ...; **end**;
...
process $P_n(...)$;
  begin ...; **region** v **do** critical section n; **end**; ...; **end**;

The construct **region** v **do** S **end** can be realized by a semaphore as follows:

var *mutex_v* : *semaphore* **initial** (1);
wait(*mutex_v*); S; *signal*(*mutex_v*);

Notice that if we use critical regions we may get deadlock as the following example shows, because at Point (1) process P_1 waits for process P_2 to exit the critical region relative to the variable w, while at Point (2) process P_2 waits for process P_1 to exit the critical region relative to the variable v.

---

var v : **shared** T_1; w : **shared** T_2;
process $P_1(...)$;
  begin ...;
        region v **do** ...; (1) **region** w **do** critical section 1; **end**; ...; **end**;
        ...;
  end;

process $P_2(...)$;
  begin ...;
        region w **do** ...; (2) **region** v **do** critical section 2; **end**; ...; **end**;
        ...;
  end;

Conditional Critical Regions

This construct was introduced in [12]. When using a semaphore, a process, say P, can test whether or not a variable is 0 and if it is so the process P is suspended. There is no way for P to test the value of two variables at the same time. To overcome this limitation we can use the following construct called *conditional critical region*:

---

var v : **shared** T; /* v is a variable of type T and
    can be shared among several processes */
region v **when** B **do** S **end**

This construct ensures that when the condition B is true, the process executes S and during the execution of S no other process may access the variable v, in particular, no other construct of the form**: region** v **when** B' **do** S' **end** can be executed by a process.

When a process P executes the construct **region** v **when** B **do** S **end**, first P gets mutually exclusive access to the variable v (thereby leaving the associated queue Qv) and then it tests whether or not the condition B is true. If B is true, then P executes S and exits the critical region. Otherwise, if the condition B is false, then the process P releases the mutually exclusive access to the variable v and enters another queue, call it QvB. When a process exits its critical region, all processes in the queue QvB are transferred to the queue Qv to allow one of them to regain mutually exclusive access to the variable v and to re-test whether or not the value of B is true. This re-testing generates a form of busy waiting.

This technique for executing conditional critical regions may determine unnecessary swaps from the queue QvB to the queue Qv and back. However, in practice, these disadvantages are compensated by the simplicity of using conditional critical regions, instead of semaphores.

When using conditional critical regions, there could be deadlock if the condition B is false for all processes.

By using conditional critical regions the program for the circular buffer of size N with producers and consumers is as follows. Our program is equivalent to the one of Section 4.5

where the two semaphores *mutex_s* and *mutex_r* have been replaced by a single semaphore *mutex*. Thus, it is never the case that the procedures *send* and *receive* are concurrently executed.

```
                                                    /* the circular buffer */
var BUFFER : shared record buffer : array [0..N−1] of message
                           in : 0..N−1 initial(0);
                           out : 0..N−1 initial(0);
                           free_cells : 0..N initial(N)
                     end;
procedure send(x : message);
  begin region BUFFER when free_cells > 0 do
        buffer[in] := x;   in := (in+1) mod N;      /* critical section 1 */
        free_cells := free_cells −1;                /* critical section 1 */
        end
  end
procedure receive(var x : message);
  begin region BUFFER when free_cells < N do
        x := buffer[out];   out := (out+1) mod N;   /* critical section 2 */
        free_cells := free_cells +1;                /* critical section 2 */
        end
  end
```

A semaphore s can be realized by using conditional critical regions as follows:

```
                    /* semaphore s realized by using conditional critical regions */
wait(s) :    region s when s>0 do s := s−1 end
signal(s) :  region s do s := s+1 end
```

Notice that **region** s **do** $s := s+1$ **end** is equivalent to
region s **when** *true* **do** $s := s+1$ **end**.

The conditional critical region **region** v **when** B **do** S **end** can be realized by using semaphores as follows [1]:

```
                    /* region v when B do S end realized by using semaphores */
var mutex_v  : semaphore initial(1);        /* mutex_v ∈ {0, 1}      */
    testB    : semaphore initial(0);        /* testB   ∈ {0, 1, ...} */
    count_B  : integer initial(0);          /* count_B ∈ {0, 1, ...} */
```

wait(*mutex_v*);
while *not B* **do**
  begin *count_B* := *count_B* + 1; *signal*(*mutex_v*);
    wait(*testB*); *wait*(*mutex_v*); **end**;
S;
while *count_B* > 0 **do**
  begin *signal*(*testB*); *count_B* := *count_B* − 1; **end**;
signal(*mutex_v*);

With reference to the queues *Qv* and *QvB* associated with the implementation of a conditional critical region, the suspended processes waiting on the semaphore *mutex_v* are inserted in the queue *Qv*, and the suspended processes waiting on the semaphore *testB* are inserted in the queue *QvB*.

Now let us illustrate in some detail how the conditional critical regions are executed by explaining the instructions of the above program and the way in which the semaphores *mutex_v* and *testB* control the activities of the processes.

Let us assume that there are several processes each of which wants to execute its own conditional critical region. Let us also assume, without loss of generality, that all conditional critical regions have the same shared variable v, that is, they are of the form: **region** v **when** B **do** S **end**, where B and S may vary from process to process. (If two conditional critical regions have different shared variables, their execution can be done in any order one desires.) Let us consider one of these processes, say P, and let us assume that it is trying to execute the conditional critical region **region** v **when** B **do** S **end**. First P gets mutually exclusive access to the variable v by performing *wait*(*mutex_v*). Then, if P finds its condition B to be false, it is suspended on the semaphore *testB*, and thus, inserted in the queue *QvB*. The number of processes which are in the queue *QvB*, is stored in the variable *count_B*. Otherwise, if the process P finds the condition B to be true, it executes S and then it executes *signal*(*testB*) as many times as the number of processes which are suspended in the queue *QvB* (that is, as many times as the value of *count_B*). Finally, the process P releases the mutually exclusive access to the variable v by performing *signal*(*mutex_v*). At this point,

(1) all processes suspended in the queue *QvB* are removed from that queue and inserted in the queue *Qv*, and then

(2) the queue *Qv* is served according to the first-in-first-out policy and thus, one process in the queue is activated. If the activated process, say Q, finds its condition B to be false, then Q releases the mutually exclusive access to the variable v (by executing *signal*(*mutex_v*)) and enters the queue *QvB* (by executing *wait*(*testB*)). Otherwise, if Q finds its condition B to be true, then Q executes S and then it executes *signal*(*testB*) as many times as the number of processes which are suspended in the queue *QvB*. Finally, the process Q releases the mutually exclusive access to the variable v by performing *signal*(*mutex_v*).

From then on, the state of affairs continues by going again through the Points (1) and (2) indicated above. The loop around these points terminates when the queues *Qv* and *QvB* are both empty.

Notice that in our program there are no statements which explicitly insert processes into queues or remove processes from the queues. These operations are implicitly performed by the *wait* and *signal* operations on the semaphores *mutex_v* and *testB*.

Notice also that, in general, the value of the condition B depends also on the value of the local variables of the individual processes. Thus, a deadlock situation may occur and, indeed, this happens when for all processes the condition B is false.

The intensive swapping of processes between the queue Qv associated with the semaphore *mutex_v* and the queue QvB associated with the semaphore *testB*, suggests the use of the conditional critical regions only for loosely connected processes, that is, processes whose interactions for accessing shared resources are not very frequent.

4.8 Monitors

In this section we examine the monitor construct which was introduced by Prof. Hoare [13] for controlling the mutual exclusive access of several processes to their critical sections.

A *monitor* is an abstract data type with local variables and procedures for reading and writing these local variables. There are no global variables. The procedures of a monitor are executed in a mutually exclusive way (see Remark 3 below). This is enforced by the system.

Here is an example of a monitor (taken from [13]) which can be used for ensuring the mutually exclusive access to a resource which is shared among several processes which dynamically take and release the resource.

```
resource : monitor;
begin free : boolean  initial(true); /* declaration and initialization of local data */
      available : condition;          /* available should not be initialized */
      entry procedure take;           /* declaration of procedures for external use */
            begin if free=false then available.wait;
                  free := false;
            end;
      entry procedure release;
            begin free := true;
                  available.signal;
            end;
end
```

The boolean variable *free* whose initial value is *true*, tells us whether or not the resource is free for use. If a process wants to take the resource and the resource is not free, then that process is delayed waiting on the variable *available* of type *condition*. This variable is signalled by a process which releases the resource.

A process which wants to take the resource should execute the procedure *take*, and a process which wants to release the resource should execute the procedure *release*. From a mathematical point of view, a monitor can also be viewed as an algebra with: (i) its data (i.e., the variables which are all local variables), (ii) the state of these data (i.e., the value of the local variables which are not condition variables), and (iii) the operations on these data.

The following remarks will clarify the notion of a monitor. The reader may also refer to [13] for more details.

Remark 1. There could be more than one reason for waiting. These reasons must all be distinguished from each other. The programmer must introduce a variable of type *condition* for each reason why a process might have to wait. Each variable of type *condition* is subject to two operations: the *wait* operation and the *signal* operation. The *wait* operation on a variable *cond* of type *condition* is denoted by *cond.wait*, and a *signal* operation on a variable *cond* of type *condition* is denoted by *cond.signal*.

A variable of type *condition* does *not* get values: it is neither *true* nor *false* and, thus, should not be initialized. We can think of a variable of type *condition* as the label which is associated with the corresponding *wait* and *signal* operations. □

Remark 2. Only *entry procedures* can be called from outside the monitor. Besides entry procedures, in a monitor there could be other *procedures*, local to the monitor. They are called by the entry procedures. The entry procedures and the other procedures may, in general, have parameters, and they are collectively called *procedures*.

Entry procedures and procedures of a monitor can only access the local variables of the same monitor. These local variables cannot be accessed from outside the monitor. □

Remark 3. We assume that inside a monitor procedure a process is either (i) in the execution state (i.e., running), or (ii) is in the waiting state (i.e., waiting) (see also Figure 11 below).

A process which has executed a *cond.wait* operation from inside a monitor procedure and has not yet resumed execution (see the following Remark 4), is *not* considered to be a process which executes that procedure. That process is on a waiting state and not in an execution state. Notice, however, when that waiting process starts the execution again, it will resume from the statement just after the *cond.wait* operation which made it to stop. Thus, it may be the case that while a process is waiting on a *cond.wait* operation on a procedure of a monitor, another process is executing the same or a distinct procedure of that same monitor.

We have the following properties.

(Mutual Exclusion) *For each monitor at any given time, there exists at most one procedure of the monitor which is executed by a process, and that procedure, if it exists, is executed by exactly one process* (This statement holds regardless whether it is relative to an entry procedure or a procedure).

(Process Locality) *For each process at any given time, there exists at most one monitor procedure among the procedures of all monitors such that the process is either executing that procedure or waiting in that procedure* (This statement holds regardless whether it is relative to an entry procedure or a procedure). □

Remark 4. A *cond.wait* operation performed on a condition variable *cond*, issued from inside a monitor procedure, causes the process which performs it, to be stopped and that process is made to wait for a future *cond.signal* operation to occur. This *cond.signal* operation will be performed by a different process. A *cond.signal* operation performed on

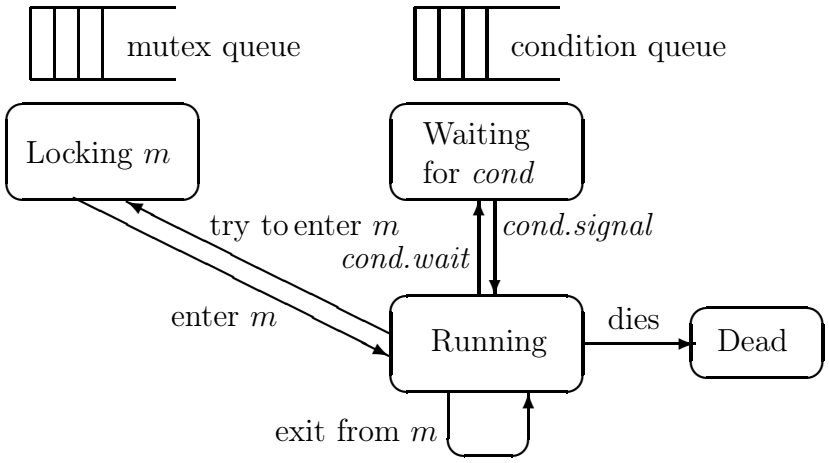


Fig. 11. Diagram of states and transitions for a process and a monitor m (according to Prof. Hoare's proposal). In general, more than one condition *cond* and more than one corresponding condition queue may be present. The *cond.signal* operation is performed by a different process.

the condition variable *cond*, issued from inside a monitor procedure, causes *exactly one of the processes waiting for a cond.signal operation to be resumed immediately*, without possibility of an intervening procedure call from yet a third process. If there are no waiting processes, the *cond.signal* operation has no effect.

We assume, according to Prof. O-J. Dahl, that when a process has performed a *signal* operation for a condition on which another process is waiting, then this signalling process immediately terminates the monitor procedure it is executing. Indeed, we assume that every *signal* operation can only occur as the last operation of an entry procedure. A different option, taken by Prof. Hoare, is to assume that the signalling process is delayed until the resumed process permits it to proceed.

The *signal* operation on a condition and the immediate resumption of a process waiting on that condition avoids busy waiting. Thus, possession of the monitor can be viewed as a privilege which is explicitly passed from the signalling process to the waiting one.

Notice that a process, say P, which performs a *cond.wait* operation from inside a monitor procedure, should relinquish the exclusive use of the monitor (see Remark 3), because otherwise no process may ever perform a *cond.signal* operation and allow the process P to resume execution. In other words, when a process P has executed a *cond.wait* operation from inside a monitor procedure, then a different process, say Q, may start the execution, or resume the execution, of a procedure of that monitor. When eventually the process P resumes the execution of the monitor procedure, it does so from exactly the same point where it started waiting, that is, it resumes the execution from the statement following the *cond.wait* operation. □

Remark 5. In the *resource* monitor given above, a process which executes the *take* procedure does not have to retest that *free* has gone *true* when it resumes execution after its

waiting, because the *release* procedure has guaranteed that this is so. Recall that we assume that no other process can intervene between a *signal* operation and the resumption of a waiting process (and only one will be resumed). □

Remark 6. The processes which are waiting on a condition are resumed according to a *first-in-first-out* policy. This ensures a simple queueing discipline so that every process waiting on a given condition will eventually get its turn, if that condition is signaled sufficiently many times. Below in this section we will see that by using a parametric waiting one can establish a different resumption policy. □

In the following example taken from [13], we consider a monitor for controlling a set of producer-consumer processes which communicate via a circular buffer, also called *bounded buffer* (see also Section 4.5). A bounded buffer is defined as a sequence of at most $N(\geq 0)$ *portions* realized by an array of N cells, from cell 0 to cell $N-1$. A producer appends a new portion at the end of the buffer in the position of the array denoted by the variable *last*, which gets values in the interval $[0..N-1]$ and it is initially 0. A consumer updates the buffer by removing its first portion which is the position $(last - count)\ mod\ N$, where *count* denotes the number of portions in the buffer. Initially the buffer is empty, that is, $count = 0$.

```
bounded_buffer : monitor;
begin buffer : array [0..N−1] of portion;
      last : 0..N−1 initial(0);
                  /* last is the buffer position where to append a new portion */
      count : 0..N initial(0);
                  /* count holds the length of the sequence, initially 0 */
      nonempty, nonfull : condition;
      entry procedure append(x : portion);
          begin if count = N then nonfull.wait          /* 0 ≤ count < N */
                                  /* wait until the buffer becomes not full */
                buffer[last] := x;
                last := (last + 1) mod N;
                count := count + 1;
                nonempty.signal
          end;
      entry procedure remove(var x : portion);
          begin if count = 0 then nonempty.wait;        /* 0 < count ≤ N */
                                  /* wait until the buffer becomes not empty */
                x := buffer[(last−count) mod N];
                count := count − 1;
                nonfull.signal
          end
end
```

In this *bounded_buffer* monitor we have indicated within comments two constraints for the value of the variable *count*. These constraints hold during program execution at the point where they are written.

Notice that the use of monitors avoids the repetition of tests of conditions which occurs if we use conditional critical regions. If we use monitors, in fact, conditions have names and they can be explicitly signalled.

Now we address the problem of defining the policy for resuming programs waiting on a condition. Let us follow the approach proposed in [13]. We introduce a *wait* operation with a parameter, say p, which indicates the priority of the waiting on a condition *cond*. This *wait* operation is as follows:

cond.wait(p);

where p is an integer expression which is evaluated at the time of the execution of *cond.wait*(p). The value of p is associated with the waiting program. After the execution of *cond.signal*, it is the program that specified the lowest value of p that is resumed.

When using this parametric *wait* operation it may happen that a program overtakes another program infinitely many times. The programmer must take care to avoid such situation which is often undesirable. It can do so by making p to be a nondecreasing function of the time at which the wait begins.

Here is an example of a monitor which uses the parametric *wait* operation. This monitor allows a program to delay itself for n units of time. In order to do so the program calls the procedure *wakeme*(n). The procedure *tick* is called by the hardware every unit of time.

```
alarmclock : monitor;
begin now : integer initial(0);
      wakeup : condition;
      entry procedure wakeme(n : integer);
          begin alarmsetting : integer;
                alarmsetting := now + n;
                while now < alarmsetting do wakeup.wait(alarmsetting);
                wakeup.signal                          /* ——— (4) */
          end;

      entry procedure tick;
          begin now := now + 1; wakeup.signal end
end
```

The *signal* operation (4) is for the next waiting program which is due to wake up at the same time.

The reader is encouraged to read the original paper [13] where monitors are introduced: it is a beautiful example of precision and clarity.

4.9 Using Monitors for Solving the Five Philosophers Problem

The *five philosophers problem*, also called the *dining philosophers problem*, was first posed by Prof. Dijkstra. It can be described as follows. There are five philosophers, denoted by the numbers 0, 1, 2, 3, and 4, sitting in the clockwise order at a table (see Figure 12). There are also five forks on the table, each one in between two philosophers. In the center of the table there is a bowl of spaghetti which is continuously refilled. Each philosopher is in the following endless cycle:

while *true* **do** think; eat **od**

Thus, each philosopher wants to think and then to eat and then to think again, and so on. It is assumed that for each philosopher the thinking period and the eating period are *finite*. In order to be able to eat each philosopher needs the two forks which are next to him, the one on his left and the one on his right.

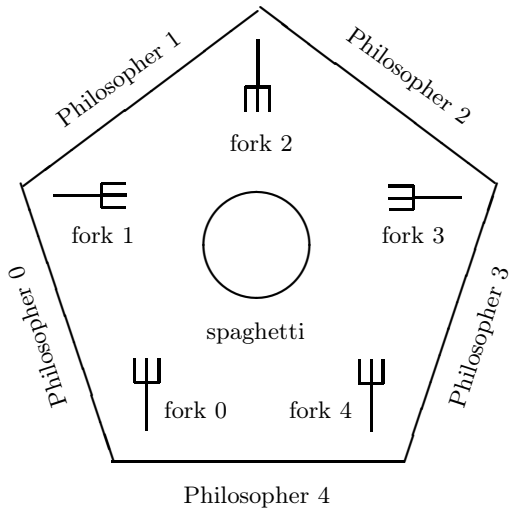


Fig. 12. The five philosophers problem.

We want to provide a mechanism to allow the following:
(i) no fork can be held by two philosophers at the same time,
(ii) no deadlock occurs, that is, for each time t there exists a philosopher i, with $0 \leq i \leq 4$, and there exists a time u after t such that at time u the philosopher i eats, and
(iii) no starvation occurs, that is, for each time t and for each philosopher i, with $0 \leq i \leq 4$, there exists a time u after t such that at time u philosopher i eats.

Let us first present a *tentative* (and incorrect!) solution which consists of:
(i) the monitor *av_fork_monitor* for controlling the access to the forks, and
(ii) for $i = 0, \ldots, 4$, the program *philosopher*(i) for the i-th philosopher [2, page 78].

For $i = 0, \ldots, 4$, *av_fork*$[i]$ is the number of available forks for the i-th philosopher and, initially, *av_fork*$[i]$ is 2. All additions and subtractions below are performed modulo 5.

```
                                        /* Five Philosophers: tentative solution */

av_fork_monitor : monitor;
begin av_fork : array [0..4] of 0..2  initial_all(2);
      two_forks : array [0..4] of condition;

      entry procedure takeforks(i : 0..4);
          begin if av_fork[i]<2 then two_forks[i].wait;
                av_fork[i+1] := av_fork[i+1] − 1;
                av_fork[i−1] := av_fork[i−1] − 1;
          end;

      entry procedure releaseforks(i : 0..4);
          begin av_fork[i+1] := av_fork[i+1] + 1;
                av_fork[i−1] := av_fork[i−1] + 1;
                if av_fork[i+1] = 2 then two_forks[i+1].signal;
                if av_fork[i−1] = 2 then two_forks[i−1].signal;
          end;
end

process philosopher(i : [0..4]);
begin while true do think; takeforks(i); eat; releaseforks(i); od end;
```

The main program is:

```
cobegin
philosopher(0); philosopher(1); philosopher(2); philosopher(3); philosopher(4);
coend
```

This tentative solution to the five philosophers problem using monitors, is *not* satisfactory (and thus, it is *not* a solution), because it allows the starvation of a philosopher if his two neighbours perform a suitable sequence of actions.

In the following Table 1 we indicated a sequence of actions which shows this possibility, so that philosopher 1 and 3 may cause the starvation of philosopher 2. (In Table 1 we have written -*n*- in the column of $av\_fork[i]$, for $i = 0, 2, 4$, to mean that the value of $av\_fork[i]$ changed from the previous unit of time, that is, the row above.) Indeed, after time 7, the subsequence:

releaseforks(1); *takeforks*(1); *releaseforks*(3); *takeforks*(3);

may be repeated forever because the forks at time 7 are exactly as at time 3. This sequence of actions can be generated by philosophers 1 and 3 which coordinate their actions on the basis, for instance, of the value of a shared variable.

action	time	*av_fork*[0]	*av_fork*[1]	*av_fork*[2]	*av_fork*[3]	*av_fork*[4]
	0 :	2	2	2	2	2
takeforks(1)	1 :	-1-	2	-1-	2	2
takeforks(3)	2 :	1	2	-0-	2	-1-
two_forks[2].*wait*	3 :	1	2	0	2	1
releaseforks(1)	4 :	-2-	2	-1-	2	1
takeforks(1)	5 :	-1-	2	-0-	2	1
releaseforks(3)	6 :	1	2	-1-	2	-2-
takeforks(3)	7 :	1	2	-0-	2	-1-

Table 1. Conspiring behaviour of philosophers 1 and 3 against philosopher 2. Rows at time 3 and 7 are equal.

The cooperation of philosophers 0 and 4 can prevent such conspiring behaviour of philosophers 1 and 3. Indeed, while philosopher 1 is eating, philosopher 4 should prevent philosopher 3 from releasing and taking forks in succession, that is, eating twice (see rows at time 6 and 7 of Table 1 below). Philosopher 4 can do so by eating once in between the two eating sessions of philosopher 3, and philosopher 4 can do so if philosopher 0 does not compete with him for a fork at that time (precisely at this point cooperation between philosopher 0 and 4 is required). While philosopher 4 is eating, philosopher 2 can eat and avoids possible starvation planned against him by philosophers 1 and 3.

In order to avoid the conspiring behaviour of philosophers 1 and 3 against philosopher 2 without relying on the cooperation between philosophers 0 and 4 or any other philosophers, we may use the following improved program for the monitor.

This program provides a *correct* solution to the five philosophers problem because: (i) mutual exclusion is ensured, (ii) deadlock is avoided, and (iii) starvation is avoided. All additions are performed modulo 5.

The tentative solution above is not correct because it allows starvation due to *bad cooperation* between some philosophers, while a correct solution should avoid starvation in *all* circumstances. In the same way, a correct solution should avoid starvation without assuming *good cooperation* between some philosophers.

```
                                        /* Five Philosophers: monitor solution */
fork_monitor : monitor;
begin freefork : array [0..4] of boolean  initial_all(true);
      availablefork : array [0..4] of condition;

      entry procedure takeforks(i : 0..4);
          begin if freefork[i] = false then availablefork[i].wait;
                    freefork[i] := false;
                if freefork[i+1] = false then availablefork[i+1].wait;
                    freefork[i+1] := false;
          end;
```

entry procedure *releaseforks*(i : 0..4);
begin *freefork*[i] := *true*;
availablefork[i].*signal*;
freefork[i+1] := *true*;
availablefork[i+1].*signal*;
end;

process *philosopher*(i : [0..4]);
begin while *true* **do** think; *takeforks*(i); eat; *releaseforks*(i); **od end**;

The main program is:

cobegin
philosopher(0); *philosopher*(1); *philosopher*(2); *philosopher*(3); *philosopher*(4);
coend

In this solution to the five philosophers problem the first philosopher who enters the monitor will succeed in taking his two forks and he will eat without any delay. Subsequently, other philosophers may be delayed, but no deadlock or starvation may occur. Also conspiring behaviour is impossible. The formal proofs of these properties are left to the reader. These proofs are based on the fact that the programs which are waiting on a condition, are resumed according to a first-in-first-out policy.

Remark 1. Notice that the monitor *fork_monitor* which provides a solution to the five philosopher problem, can be viewed as the generalization 'to five dimensions' of the monitor *resource* presented at the beginning of Section 4.8. Indeed, *free* : *boolean* is generalized to the array *freefork* : **array** [0..4] **of** *boolean*, and *available* : *condition* is generalized to the array *availablefork* : **array** [0..4] **of** *condition*. □

Remark 2. In the above monitor solution the level of concurrency is *not* maximal. Indeed, for instance, the fact that there is one monitor implies that no two processes may at the same time release forks. Obviously, the concurrent execution of these actions could have been allowed without destroying any property of interest such as the absence of deadlock or the absence of starvation. Likewise, in our monitor solution we have sequentialized other actions even if there was no real need. For instance, the two **if-then** statements inside the *takeforks* procedure are performed one after the other, while one could have also allowed their parallel execution. □

4.10 Using Semaphores for Solving the Five Philosophers Problem

In this section we present a solution to the five philosophers problem using semaphores. Deadlock and starvation are avoided. In particular, one can show that no conspiring behaviour can be realized even in the presence of malicious philosophers, so to speak.

In this solution *fork* is an array of semaphores. Initially, for $i = 0, \ldots, 4$, *fork*[i] is 1 and it denotes that *fork*[i] is available to *philosopher*[i] and *philosopher*[i+1] (see clause **initial_all**(1)). All additions are performed modulo 5. For $i = 0, \ldots, 4$, *philosopher*[i] needs *fork*[i] and *fork*[i+1] to eat. The extra counting semaphore *room* allows at most 4 philosophers to enter the dining room. It is easy to see that if there are at most 4 philosophers

sitting at the table then one of them, say *philosopher*[i], has both *fork*[i] and *fork*[$i+1$] available. Thus, initially, deadlock is impossible. After eating every philosopher has to leave the room, i.e., he performs *signal*(*room*), and thus, deadlock is always impossible.

If we assume that processes waiting on the *room* semaphore and any *fork*[i] semaphore, for $i = 0, \ldots, 4$, are served according to the first-in-first-out policy, then deadlock and starvation are impossible and, in particular, conspiring behaviour against a particular philosopher is impossible. We leave the formal proof of these properties to the reader.

```
                                        /* Five Philosophers: semaphore solution */
program five_philosophers;
var fork : array [0..4] of semaphore initial_all(1);      /* array [0..4] of 0..1 */
    room : semaphore initial(4);                          /* room ∈ {0,1,2,3,4} */
process philosopher(i : [0..4]);
begin while true do
    think;                                                /* non-critical section */
    wait(room);
    wait(fork[i]); wait(fork[i+1]);
    eat;                                                  /* critical section */
    signal(fork[i]); signal(fork[i+1]);
    signal(room) od
end;
cobegin
philosopher(0); philosopher(1); philosopher(2); philosopher(3); philosopher(4);
coend
```

In the following Points (1) and (2) we briefly compare the semaphore technique for solving the five philosophers problem with the monitor technique we have presented in the previous Section 4.9. Indeed, this comparison is based on properties which hold for the semaphore and monitor techniques in general, not only for the application of these techniques to the solution to the five philosophers problem.

Point (1). In order to achieve the mutually exclusive execution of the critical sections, that is, for performing *the sequences of actions* relative to the critical sections in an atomic way, the semaphore technique relies on the *atomicity* of the execution of each assignment and test, and this atomicity is ensured by the hardware. Also monitors achieve mutual exclusion by relying on the atomicity of each assignment and test.

Point (2). Semaphores do not adopt a fixed policy for scheduling the waiting processes. On the contrary, monitors assume a fixed *queue* policy, that is, the first-in-first-out policy. □

In practice, in order to ensure mutual exclusion, people have given preference to the use of monitors with respect to that of semaphores, or critical regions, or conditional critical regions. This is because monitors offer the benefit of structuring data and operations in a single construct that realizes what in mathematical terms is called an algebra.

Let us end this section by listing some general properties a concurrent solution to a given problem should enjoy and, in particular, these properties are enjoyed by the monitor solution to the five philosophers problem we have presented in Section 4.9.

(i) The solution should be *distributed*, that is, without a 'master process' which schedules the actions of all other processes and has knowledge of all their activities.
(ii) The solution should be *highly concurrent*, that is, it should allow as many processes as possible to operate at the same time, without forcing unnecessary sequentiality among them.
(iii) The solution should be *easily programmable* in the given programming language.
(iv) The *correctness* of the solution should be *easily provable*.

All these requirements have motivated the various proposals one can find in the literature for the solution of the five philosophers problem. In particular, the monitor solution: (i) is distributed at the level of a monitor procedure, that is, one process at a time may execute a procedure of the monitor (that is, each procedure is executed atomically), (ii) is highly concurrent because the only queueing mechanism used is the one for scheduling the processes waiting on a condition, (iii) is easily programmable, and (iv) can easily be proved correct as it has been demonstrated in practice.

As we have seen in the case of the five philosophers problem, when looking for a concurrent solution to a given problem, we often want, besides the mutual exclusion property, other properties to hold. In particular, we may want the absence of deadlock and the absence of starvation.

In the following section the reader will see an elegant solution to a concurrent problem which indeed guarantees mutual exclusion, absence of deadlock, and absence of starvation. However, these properties will be guaranteed through the use of shared variables, not monitors.

4.11 Peterson's Algorithm for Mutual Exclusion

Let us consider two processes P_1 and P_2 running in parallel, each of which is of the form:

An abstract process for Peterson's algorithm: form (#1)

while *true* **do**
    non-critical section;
    critical section;
        od

We assume that for each process the non-critical section and the critical section have *finite* duration. We want to ensure mutual exclusion, in the sense that at any time at most one of process can be in its critical section. This property can be achieved by using the following protocol, called Peterson's algorithm (see also [15]). Peterson's algorithm ensures mutual exclusion at the expense of adding some instructions before entering the critical section (these instructions are also called *pre-protocol* or *entry-protocol*) and some instructions after exiting the critical section (these instructions are also called *post-protocol* or *exit-protocol*). These instructions refer to variables which are shared between the two processes P_1 and P_2. The presence of at least a shared variable is necessary because otherwise no mutual exclusion can be guaranteed.

Peterson's Algorithm for 2 processes (Version A)

$q_1 := \mathit{false}; \quad q_2 := \mathit{false}; \quad s := 1;$

P_1 : **while** *true* **do**	P_2 : **while** *true* **do**
l_1 : non-critical section 1;	m_1 : non-critical section 2;
l_2 : $q_1 := \mathit{true}$; $s := 1$;	m_2 : $q_2 := \mathit{true}$; $s := 2$;
l_3 : **await** $(\neg q_2) \vee (s = 2)$;	m_3 : **await** $(\neg q_1) \vee (s = 1)$;
l_4 : critical section 1;	m_4 : critical section 2;
l_5 : $q_1 := \mathit{false}$; **od**	m_5 : $q_2 := \mathit{false}$; **od**

The vertical bar between the two processes indicates that they are supposed to run concurrently. The variables q_1, q_2, and s are shared between the two processes. The variable q_1 can be written by process P_1 only, the variable q_2 can be written by process P_2 only, and the variable s can be written by both P_1 and P_2. The existence of the shared variable s which is written by the two processes is a limitation of Peterson's algorithm. There exist other algorithms which ensure mutual exclusion and whose variables are not written by more than one process. One of these algorithms is the Bakery protocol due to L. Lamport [14]. This algorithm, however, requires a variable which can take an unbounded integer value.

In Peterson's algorithm the statement '**await** *cond*', where *cond* is a boolean expression, stands for the following statement:

l : **if** *cond* **then** *skip* **else** *goto* l

Thus, the execution of the statement '**await** *cond*' is equivalent to the execution of a sequence of one or more statements of the form '**if** *cond* **then** *skip* **else** *goto* l'.

During the execution of that sequence of **if-then-else** statements all evaluations of the *cond* expression return the value *false*, except the last one which returns the value *true*.

Notice that Peterson's algorithm allows that, while a process is in its non-critical section, the other process may execute its critical section as many times as desired. That behaviour is obtained by using the variables q_1 and q_2: this will be clear from the proof of correctness of the protocol which we will give below in Section 6.2. The variable s is required for keeping, so to speak, the value of the turn between the two processes when they desire to enter their critical sections. The initialization of s to 1 or 2 is irrelevant.

An informal explanation of how Peterson's algorithm works and why it ensures mutual exclusion is as follows. For this explanation we assume that the two sequences of the statements, the one at label l_2 and the one at label m_2, are both *atomic* (see Definition 1 at page 17). We will say that at time t process P_1 (or P_2) has *made a request* to enter its critical section iff at time t it has completed the execution of the two statements at label l_2 (or m_2, respectively). Now let us consider process P_1 and a generic execution of the body of its **while-do** statement. If during that execution at time t, process P_1 has made a new request to enter its critical section, it may enter its critical section at time $\bar{t} > t$ iff $(\neg q_2) \vee (s = 2)$ is *true* at time $\bar{t}$, that is, *either*
(Case $\neg q_2$) during the time interval $(t, \bar{t}\,]$ process P_2 executed the statement q_2 :=*false* and did not make any new request to enter its critical section, *or*
(Case $s = 2$) during the time interval $(t, \bar{t}\,]$ process P_2 made a new request to enter its critical section and, thus, process P_2 made its request after process P_1.

Analogous argument holds for process P_2, instead of process P_1, by interchanging 1 and 2.

Many interesting properties of Peterson's algorithm, as it is the case of many other protocols, depend on the atomicity of the statements or sequences of statements. Now we specify that atomicity.

First of all, it is obvious that in Peterson's algorithm the execution of the entire sequence of the **if-then-else** statements which corresponds to the '**await** *cond*' statement, is *not* atomic because, otherwise, no process can change the value of the condition *cond* and, if a process has to wait before entering its critical section, it will have to wait forever. Indeed, we require that only the execution of each statement of the form

if *cond* **then** *skip* **else** *goto* l

is atomic. This requirement is equivalent to the requirement that the evaluation of the condition *cond* of each '**if** *cond* **then** *skip* **else** *goto* l' statement is atomic.

Each execution of an assignment statement is atomic, while the execution of the sequence of the two statements '$q_1 := true$; $s := 1$;' need *not* be atomic. This means that mutual exclusion continues to hold if we allow the execution of one or more statements of process P_2 after the execution of $q_1 := true$ and before the execution of $s := 1$. We will express this fact by writing, according to [15]:

l_{21} : $q_1 := true$;
l_{22} : $s := 1$;

instead of:

l_2 : $q_1 := true$; $s := 1$;

(see the Version A1 and the Version B of Peterson's algorithm below). Analogously, mutual exclusion continues to hold if the execution of the sequence of the two statements '$q_2 := true$; $s := 2$;' is not atomic.

As already mentioned, in Peterson's algorithm the variable s can be set and read by both processes. This is a weakness of the algorithm and may create problems, not for ensuring the mutual exclusion property, but for ensuring a property, called *starvation*, as we now indicate.

Let us first define the notions of *starvation* and *failure* in the case of Peterson's algorithm and then we will see that, unfortunately, a process which executes Peterson's algorithm may starve due to the failure of another process.

Definition 3. (i) [*Starvation*] We say that a process *starves* iff it does not complete the execution of an **await** statement, that is, during the execution of '**await** *cond*' the process always finds the value of *cond* to be false.

(ii) [*Failure*] We say that a process *fails after statement st* occurring in the body of the **while-do** iff after executing the statement *st*, it does not execute any more statements. We say that a process *fails* iff there exists a statement *st* in the body of the **while-do** such that it fails after *st*.

We assume that a process which *atomically* executes a sequence '$st_1; \ldots; st_n$;' of statements, with $n > 1$, does *not* fail after any of the statements $st_1, \ldots, st_{n-1}$. If that process fails during the execution of that sequence, it fails after the statement st_n.

Now if we allow a process to fail when executing Peterson's algorithm, then the following unfortunate situation may arise: a process, say P_1, is at label l_3 and the other process P_2 fails after the statement $q_2 := true$ and never sets s to 2. In this situation process P_1 starves and never enters its critical section.

If we assume the atomicity of the two sequences of statements: '$q_1 := true$; $s := 1$;' and '$q_2 := true$; $s := 2$;' then Peterson's algorithm ensures that a process does not starve even if the other process fails after any one of the following four statements: (i) $s := 1$ (for process P_1), (ii) $q_1 := false$ (for process P_1), (iii) $s := 2$ (for process P_2), and (iv) $q_2 := false$ (for process P_2). With these atomicity requirements Peterson's algorithm also ensures that a process does not starve even if the other process fails after a statement occurring in the non-critical section. (However, we do not allow this kind of failures because we assume that the duration of every non-critical section is finite.)

Even if we assume the atomicity of the two sequences '$q_1 := true$; $s := 1$;' and '$q_2 := true$; $s := 2$;' then Peterson's algorithm does not ensure that a process does not starve if the other process fails after the **await** *cond* statement (that is, after it has found *cond* to be true) or after a statement of the critical section. (However, we do not allow any failure within a critical section because we assume that the duration of every critical section is finite.)

From now on, unless otherwise specified, we assume that:

(i) *processes do not fail*, and
(ii) the critical and non-critical sections have *finite* duration.

The use of Peterson's algorithm ensures various properties such as mutual exclusion, absence of deadlock, absence of starvation, and bounded overtaking. Now we will formally define these properties in the general case, when we consider a set $\mathcal{P}$ of processes of the following form (#2) (which is a generalization of the form (#1) introduced at the beginning of this Section 4.11):

An abstract process for Peterson's algorithm: form (#2)

while *true* **do**
non-critical section;
waiting section (here the process waits to enter its critical section);
critical section;
od

Processes of this form (#2), besides the critical and non-critical sections, have also an intermediate section, called *waiting section*, where they wait to enter their critical section. Notice that we do *not* assume that the duration of the waiting section is finite.

Now we define the four properties that are guaranteed by Peterson's algorithm.

Definition 4. (i) *Mutual exclusion* holds for $\mathcal{P}$ iff for each time t there exists at most one process in $\mathcal{P}$ which is in its critical section at time t.
(ii) *Absence of deadlock* holds for $\mathcal{P}$ iff for each time t there exists a process $P \in \mathcal{P}$ and there exists a time u after t such that at time u process P is in its critical section.
(iii) *Absence of starvation* holds for $\mathcal{P}$ iff for each time t and for each process $P \in \mathcal{P}$ there exists a time u after t such that at time u process P is in its critical section.
(iv) *Bounded overtaking of degree k* holds for $\mathcal{P}$ iff for any process $P \in \mathcal{P}$, while P is in its waiting section, any other process $Q \in \mathcal{P}$ goes from its critical section to its non-critical section at most k times. Bounded overtaking of degree 1 is simply called *bounded overtaking*.

Remark 1. Since we assume that processes do not fail and for each process the non-critical section and the critical section have finite duration, we have that the bounded overtaking property together with the absence of deadlock implies the absence of starvation. □

Remark 2. With reference to Peterson's algorithm, absence of starvation can be defined as the conjunction of the following two statements:
(i) process P_1 is at label l_3 only a finite amount of time (and then it will enter its critical section at label l_4) and, symmetrically,
(ii) process P_2 is at label m_3 only a finite amount of time (and then it will enter its critical section at label m_4). □

Remark 3. With reference to Peterson's algorithm and the definition of the bounded overtaking property (see Definition 4 above), the sentence: 'process P_1 is in its waiting section' should be understood in the sense that process P_1 is in the interval of time after

the execution of both statements at label l_2 and it has not yet performed any statement of the immediately subsequent critical section at label l_4, that is, process P_1 is in the interval of time in which it executes the sequence of atomic statements of the form: '**if** $(\neg q_2) \vee (s = 2)$ **then** *skip* **else** *goto* l_3' which corresponds to the statement '**await** $(\neg q_2) \vee (s = 2)$'.

Analogously, the sentence: 'process P_2 is in its waiting section' should be understood in the sense that process P_2 is in the interval of time after the execution of both statements at label m_2 and it has not yet performed any statement of the immediately subsequent critical section at label m_4, that is, process P_2 is in the interval of time in which it executes the sequence of atomic statements of the form: '**if** $(\neg q_1) \vee (s = 1)$ **then** *skip* **else** *goto* m_3' which corresponds to the statement '**await** $(\neg q_1) \vee (s = 1)$'. □

Remark 4. The sentences 'process P_1 is in its waiting section' and 'process P_2 is in its waiting section' should be understood in the sense of the previous Remark 3 also in the case when we consider a variant of Peterson's algorithm where the two assignments at label l_2 and m_2 are interchanged, that is, at label l_2 we have '$s := 1$; $q_1 := \mathit{true}$;' and at label m_2 we have '$s := 2$; $q_2 := \mathit{true}$;'.

Indeed, for instance, during the interval of time in which process P_1 has executed the assignment $s := 1$ and it has not yet executed the assignment $q_1 := \mathit{true}$, process P_2 may execute its critical section an unbounded number of times because q_1 is *false*.

That variant of Peterson's algorithm is not good, because it does *not* ensure the mutual exclusion property (see below). □

Let us consider Peterson's algorithm in which each assignment to the variables q_1, q_2, and s is atomic, and the two tests '$\neg q_2 \vee s = 2$' and '$\neg q_1 \vee s = 1$' are atomic. Thus, we write Peterson's algorithm as follows:

Peterson's Algorithm for 2 processes (Version A1: finer atomicity)

$q_1 := \mathit{false}$; $q_2 := \mathit{false}$; $s := 1$;

P_1 : **while** *true* **do**	P_2 : **while** *true* **do**
l_1 : non-critical section 1;	m_1 : non-critical section 2;
l_{21} : $q_1 := \mathit{true}$;	m_{21} : $q_2 := \mathit{true}$;
l_{22} : $s := 1$;	m_{22} : $s := 2$;
l_3 : **if** $\neg q_2 \vee s = 2$ **then** *goto* l_4 **else** *goto* l_3;	m_3 : **if** $\neg q_1 \vee s = 1$ **then** *goto* m_4 **else** *goto* m_3;
l_4 : critical section 1;	m_4 : critical section 2;
l_5 : $q_1 := \mathit{false}$; **od**	m_5 : $q_2 := \mathit{false}$; **od**

---

where the execution of the statements at each label is atomic. Thus, in particular, the execution of each **if-then-else** statement is atomic.

Now we show that this version of Peterson's algorithm guarantees mutual exclusion, absence of deadlock, absence of starvation, and bounded overtaking of degree 1.

Note that the mutual exclusion property does *not* hold if we interchange the statement at label l_{21} with that at label l_{22} and/or the statement at label m_{21} with that at label m_{22}.

In particular, if we make both of these interchanges and we start from initial state where $q_1 = q_2 = \mathit{false}$ and $s = 1$, both processes P_1 and P_2 may be in their critical section at the same time, as shown by the following sequence of statements (each of which is performed by the process indicated between parentheses):

$(P1)$ $s := 1$;
$(P2)$ $s := 2$;
$(P2)$ $q_2 := \mathit{true}$;
$(P2)$ **if** $\neg q_1 \vee s = 1$ **then** *goto* m_4 **else** *goto* m_3; (thus, $P2$ enters its critical section 2)
$(P1)$ $q_1 := \mathit{true}$;
$(P1)$ **if** $\neg q_2 \vee s = 2$ **then** *goto* l_4 **else** *goto* l_3; (thus, $P1$ enters its critical section 1).

Mutual Exclusion. In order to show that Peterson's algorithm ensures mutual exclusion it is enough to show that it is never the case that at the same time process P_1 is at label l_4 and process P_2 is at label m_4. The proof of this property is by absurdum.

Let us assume that both P_1 and P_2 are at the same time in their critical section. Thus, it should be the case that $((\neg q_2) \vee (s = 2)) \wedge ((\neg q_1) \vee (s = 1))$. We have that $q_1 = q_2 = \mathit{true}$, and therefore, it should be the case that $(s = 2) \wedge (s = 1)$. But this is impossible. Hence, only one process passed its test at the **await** statement, and the second process did some assignments before passing its test, but this cannot happens either, because the last assignment performed by the second process before the test, sets s to a value which makes it impossible for it to pass its test. □

Absence of Deadlock and Absence of Starvation. In order to show that Peterson's algorithm ensures absence of deadlock it is enough to show that if process P_1 is at label l_3 and at the same time process P_2 is at label m_3, then after a finite amount of time, either process P_1 is at label l_4 or process P_2 is at label m_4. Indeed, we will show a stronger property, that is, absence of starvation. In order to show absence of starvation it is enough to show that:
(iii.1) if process P_1 is at label l_3 then after a finite amount of time, process P_1 is at label l_4 and, symmetrically,
(iii.2) if process P_2 is at label m_3 then after a finite amount of time, process P_2 is at label m_4.

Let us first show Point (iii.1). At label l_3 process P_1 repeatedly evaluates the expression $(\neg q_2) \vee (s = 2)$. While process P_1 evaluates that expression, after a finite amount of time, since process P_2 does not fail, one of the following three situations occurs:
(S1) P_2 is at label m_1: in this case P_1 finds q_2 to be *false* and P_1 will enter its critical section and it will be at label l_4;
(S2) P_2 is at label m_3: in this case either P_1 or P_2 will enter its own critical section because $s = 1$ or $s = 2$. If P_1 enters its critical section we have shown that P_1 indeed proceeds. If P_2 enters its critical section, after a finite time it will set q_2 to *false* and P_1 will then enter its critical section (see Situation S1).
(S3) P_2 is at label m_j with $j = 2$ or $j = 4$ or $j = 5$: in this case P_2 will leave this label in a finite time and process P_1 will proceed because in a finite time either Situation (S1) or Situation (S2) will occur.

The proof of Point (iii.2) is analogous to that of Point (iii.1). □

Bounded Overtaking. In order to show that Peterson's algorithm ensures bounded overtaking (that is, bounded overtaking of degree 1) it is enough to show that:
(iv.1) while process P_1 is at label l_3, process P_2 is *at most once* at label m_4 and, symmetrically,
(iv.2) while process P_2 is at label m_3, process P_1 is *at most once* at label l_4.
In other words, Point (iv.1) (and, symmetrically, Point (iv.2)) tells us that while process P_1 is waiting for entering its critical section, process P_2 cannot complete more than once its critical section, that is, P_2 can pass from label m_4 to label m_5 at most once.

Let us show Point (iv.1). We have that after setting the value of s, process P_1 has to wait at label m_3 the completion of at most one execution of the critical section by the other process P_2, because at the end of its critical section, P_2 sets the value of q_2 to *false* (thus, at this time process P_1 may enter its critical section) and if process P_2 tries to enter its critical section a second time, P_2 sets the value of s to 2, thereby giving the turn to P_1.

The proof of Point (iv.2) is analogous to that of Point (iv.1). □

We leave to the reader to show that mutual exclusion, absence of deadlock, absence of starvation, and bounded overtaking of degree 1 also hold for Peterson's algorithm if we assume the following properties:
(i) processes do not fail,
(ii) for each process the non-critical section and the critical section have *finite* duration, and
(iii) each assignment and test of the variables q_1, q_2, and s is atomic. Thus, in particular, we do not assume that the two tests '$\neg q_2 \vee s=2$' and '$\neg q_1 \vee s=1$' are atomic.

In order to denote these atomicity assumptions, when we write Peterson's algorithm, we replace the statement:

l_3 : **await** $(\neg q_2) \vee (s=2)$;
l_4 : ...

by the following two statements:

l_{31} : **if** $\neg q_2$ **then** *goto* l_4 **else** *goto* l_{32};
l_{32} : **if** $s=2$ **then** *goto* l_4 **else** *goto* l_{31};
l_4 : ...

where the execution of each **if-then-else** statement is atomic. Thus, one or more statements of process P_2 may be executed when process P_1 has finished the evaluation of $\neg q_2$ and not yet begun the evaluation of $s=2$, or it has finished the evaluation of $s=2$ and not yet begun the evaluation of $\neg q_2$.

Analogously, we replace the statement:

m_3 : **await** $(\neg q_1) \vee (s=1)$;
m_4 : ...

by the following two statements:

m_{31} : **if** $\neg q_1$ **then** *goto* m_4 **else** *goto* m_{32};
m_{32} : **if** $s=1$ **then** *goto* m_4 **else** *goto* m_{31};
m_4 : ...

where the execution of each **if-then-else** statement is atomic.

We get the following version of Peterson's algorithm:

Peterson's Algorithm for 2 processes (Version B)

$q_1 := \mathit{false}; \quad q_2 := \mathit{false}; \quad s := 1;$

P_1 : **while** *true* **do**	P_2 : **while** *true* **do**
l_1 : non-critical section 1;	m_1 : non-critical section 2;
l_{21} : $q_1 := \mathit{true}$;	m_{21} : $q_2 := \mathit{true}$;
l_{22} : $s := 1$;	m_{22} : $s := 2$;
l_{31} : **if** $\neg q_2$ **then** *goto* l_4 **else** *goto* l_{32};	m_{31} : **if** $\neg q_1$ **then** *goto* m_4 **else** *goto* m_{32};
l_{32} : **if** $s=2$ **then** *goto* l_4 **else** *goto* l_{31};	m_{32} : **if** $s=1$ **then** *goto* m_4 **else** *goto* m_{31};
l_4 : critical section 1;	m_4 : critical section 2;
l_5 : $q_1 := \mathit{false}$; **od**	m_5 : $q_2 := \mathit{false}$; **od**

where the execution of the statements at each label is atomic. Thus, in particular, the execution of each **if-then-else** statement is atomic.

Peterson's algorithm for two processes can be generalized to the case of $n\,(>2)$ processes. We can explain this n process algorithm by saying that the two process algorithm presented above is used repeatedly 'for $n-1$ levels'. By doing so we eliminate at least one process per level, until only one process remains and this winning process enters its critical section. Before performing this generalization, the two process version of Peterson's algorithm is modified by replacing the condition '$s = 2$' by the equivalent condition '$s \neq 1$' and, analogously, the condition '$s = 1$' by the equivalent condition '$s \neq 2$'. We also replace *false* by 0 and *true* by 1, and we replace the tests $\neg q_1$ by $q_1 < 1$ and $\neg q_2$ by $q_2 < 1$.

In the case of $n\,(>2)$ processes Peterson's algorithm requires two shared arrays:
- an array $Q[1..n]$ whose n components are initially set to 0 and may get values from 0 to $n-1$, and
- an array $S[1..n-1]$ whose $n-1$ components are initially set to 1 and may get values from 1 to n.

Each process P_i, for $i = 1, \ldots, n$, is of the form:

The abstract process P_i for Peterson's algorithm for n processes (with $i = 1, \ldots, n$)

while *true* **do**
l_1 : non-critical section i;
l_2 : **for** $j = 1, \ldots, n-1$ **do** $Q[i] := j; \quad S[j] := i;$
l_3 : **await** $(\forall k.\, k \neq i \rightarrow Q[k] < j) \ \vee \ (S[j] \neq i)$ **od**;
l_4 : critical section i;
l_5 : $Q[i] := 0$; **od**

The variables i (i.e., the process number), j (i.e., the index for process i), and n (i.e., the total number of processes) are local to process P_i. The test '$S[j] \neq i$' at statement l_3 is atomic and, for $k \neq i$, each test '$Q[k] < j$' is atomic. However, the whole test

'$(\forall k.\, k \neq i \;\rightarrow\; Q[k] < j) \;\vee\; (S[j] \neq i)$'
need *not* be atomic.

We leave it to the reader to check that Peterson's algorithm for n processes is correct, in the sense that, if processes do not fail, it guarantees mutual exclusion and absence of deadlock. The correctness proof of the n process version of Peterson's algorithm is a straightforward generalization of the two process proof we presented above.

Notice, however, that Peterson's algorithm for n processes guarantees neither absence of starvation nor bounded overtaking of degree k, for any $k \geq 1$, even if:
(i) processes do *not* fail, and
(ii) for every process P_i, with $i = 1, \ldots, n$, and for every $j = 1, \ldots, n-1$ we assume:
(ii.1) the atomicity of the execution of the sequence '$Q[i] := j;\; S[j] := i;$' of statements, and
(ii.2) the atomicity of the evaluation of the entire condition:
$(\forall k.\, k \neq i \;\rightarrow\; Q[k] < j) \;\vee\; (S[j] \neq i)$.
In particular, in the case of three processes P_1, P_2, and P_3, process P_1 and process P_2 may alternatively enter infinitely many times their critical sections, while process P_3 keeps on testing the condition of its topmost **await** statement (see the program below).

In the case of three processes Peterson's algorithm is as follows (for simplicity, we write Q_i instead of $Q[i]$, and S_i instead of $S[i]$):

Peterson's Algorithm for 3 processes

$$Q_1 := 0;\;\; Q_2 := 0;\;\; Q_3 := 0;\;\; S_1 := 1;\; S_2 := 1;$$

P_1 : **while** *true* **do**	P_2 : **while** *true* **do**	P_3 : **while** *true* **do**
non-critical section 1;	non-critical section 2;	non-critical section 3;
$Q_1 := 1; \quad S_1 := 1;$	$Q_2 := 1; \quad S_1 := 2;$	$Q_3 := 1; \quad S_1 := 3;$
await $(Q_2 < 1 \wedge Q_3 < 1) \vee (S_1 \neq 1);$	**await** $(Q_1 < 1 \wedge Q_3 < 1) \vee (S_1 \neq 2);$	**await** $(Q_1 < 1 \wedge Q_2 < 1) \vee (S_1 \neq 3);$
$Q_1 := 2; \quad S_2 := 1;$	$Q_2 := 2; \quad S_2 := 2;$	$Q_3 := 2; \quad S_2 := 3;$
await $(Q_2 < 2 \wedge Q_3 < 2) \vee (S_2 \neq 1);$	**await** $(Q_1 < 2 \wedge Q_3 < 2) \vee (S_2 \neq 2);$	**await** $(Q_1 < 2 \wedge Q_2 < 2) \vee (S_2 \neq 3);$
critical section 1;	critical section 2;	critical section 3;
$Q_1 := 0; \quad$ **od**	$Q_2 := 0; \quad$ **od**	$Q_3 := 0; \quad$ **od**

In this section we have considered the problem of ensuring mutual exclusion among several processes and its solution via Peterson's algorithm. Later on in Section 4.13 we will consider a different problem also concerning the achievement of a global behaviour of a set of processes which run in a concurrent way. It is the problem of detecting the termination of all the activities of a collection of processes, each of which
(i) is located at a node of a given graph,
(ii) may perform some computations, and
(iii) may delegate some computations to processes located at neighbouring nodes.

In order to illustrate a solution to this problem, we first need to present an algorithm for computing in a distributed way a spanning tree of a given finite, undirected, connected graph. This algorithm is presented in the following Section 4.12.

4.12 Distributed Computation of Spanning Trees

Let us consider a *finite, undirected, connected* graph $G = \langle N, E\rangle$ without self-loops. In particular, we assume that: (i) the set N of nodes is finite and $|N|>1$, (ii) for every arc $\langle n, m\rangle \in E$, there exists in E the arc $\langle m, n\rangle$, (iii) for every pair of distinct nodes h and k there exists a sequence $\langle n_1, n_2\rangle$, $\langle n_2, n_3\rangle$, ..., $\langle n_{p-1}, n_p\rangle$ of one or more arcs such that $n_1 = h$, and $n_p = k$, and (iv) for every node $n \in N$, $\langle n, n\rangle \notin E$. For every node $n \in N$ we define the two sets $P(n) = \{p \mid \langle p, n\rangle \in E\}$ and $S(n) = \{s \mid \langle n, s\rangle \in E\}$, which are the sets of the so called *predecessor nodes* of n and *successor nodes* of n, respectively.

We want to construct a spanning tree T of the graph G with a given node n_0 as its root by using a distributed algorithm. During the execution of the algorithm a node may be either *unmarked*, denoted $\square$, or *marked*, denoted $\boxtimes$. We assume that, initially, all nodes are unmarked.

The distributed algorithm for computing the desired spanning tree T consists of a single initial application of the rule $R1$ (see Figure 13 on the following page) to the root node n_0, followed by applications of the rules $R2$ and $R3$ (see Figure 13). Each rule is applied to a node of the graph G as follows: if the left-hand-side of a rule depicted in that figure holds at a node, then the rule fires (that is, is applied to that node) and the right-hand-side of that rule is realized at that node. The rules $R2$ and $R3$ may be applied concurrently (and in this sense the algorithm is distributed), but they should be applied in an *atomic* way, that is, when one of them is applied to a node n, no rule can be applied to a node in $P(n) \cup S(n)$.

Note that, since the graph G is modified when a rule is applied, the values of the sets $P(n)$ and $S(n)$ which are needed for applying the next rule, should be computed in the graph obtained after the application of the current rule.

Rule $R1$ is applied to the root node n_0: it erases all incoming arcs, marks the root, and sends a *mark-token*, denoted $\otimes$, to every node which is a successor of the root.

Rule $R2$ is applied to every non-root node iff there is at least an incoming mark-token from a predecessor node. When applied to node n with a mark-token arriving from node i, rule $R2$ erases all arcs incoming from nodes different from i, erases all mark-tokens arrived at node n, marks the node n, and sends a mark-token to every node which is a successor of n.

Rule $R3$ is applied to every marked, non-root node iff *all* its successor nodes have sent to that node an *end-token*, denoted $\blacktriangle$ (thus, the end-tokens travel in the opposite direction w.r.t. the direction of the arcs). In particular, rule $R3$ is applied to every marked leaf node of the spanning tree T, because every leaf node has no successors. When rule $R3$ is applied to a node n, it erases all end-tokens arrived at node n and sends an end-token to the node which is the predecessor of n. By construction, it is the case that the node n to which the rule $R3$ is applied, has one predecessor only.

One can show that in order to ensure the termination of the algorithm (see below), it is irrelevant whether or not rule $R3$ actually erases the end-tokens sent by the successor nodes.

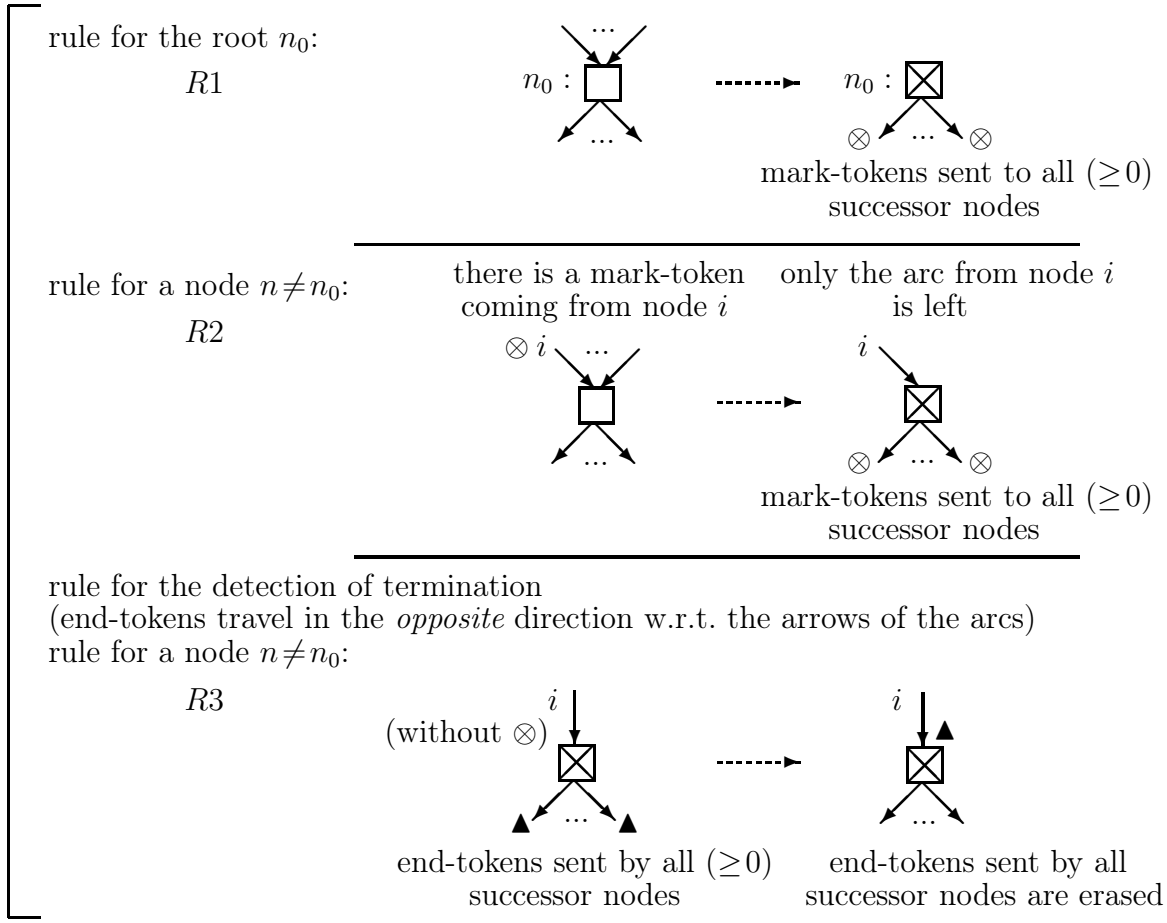


Fig. 13. Rules $R1$, $R2$, and $R3$ for the distributed computation of a spanning tree with root n_0, for any given finite, undirected, connected graph without self-loops.

Rules $R1$, $R2$, and $R3$ generate an expanding wave of mark-tokens $\otimes$ moving from the root node n_0 outward to the leaves, and then a contracting wave of end-tokens $\blacktriangle$ moving from the leaves to the root. This contracting wave is needed for detecting the termination of the algorithm.

In what follows we will assume that for every execution of our distributed algorithm which makes $k\,(\geq 0)$ rule applications (some of these applications may be made in a concurrent way in different nodes of the graph), there exists an execution of a *sequential algorithm* which makes a sequence of those k rule applications in the set $R1$; $(R2 + R3)^*$ and performs the same graph transformation. That sequence of k rule applications is said to be a *linearization* of the given execution of the distributed algorithm. An equivalent way of stating our assumption is to say that: (i) every execution of our distributed algorithm can be viewed as a *partial order* of rule applications where a rule application a_i is related to a rule application a_j iff a_i occurs before a_j (thus, the concurrent applications of the rules define the elements which are unrelated in the partial order), and (ii) the concurrent applications of the rules have an *interleaving semantics*. We also assume that for every

execution of the distributed algorithm, every linearization of that execution performs the same graph transformation.

We say that the *Termination Condition* holds for a sequence σ of rule applications if σ has a finite (proper or not) prefix which leads to a situation where every successor node of the root has sent an end-token ▲ to the root. We say that the algorithm *terminates* (or *termination occurs*) iff the linearizations of all possible executions of the algorithm enjoy the Termination Condition.

In general, a sequence of rule application may lead to a situation where no rule can fire and termination does not occur. In that case we say that there is *deadlock*.

Termination of the algorithm is ensured if we assume the following hypothesis for every sequence of rule applications in the set $R1;\ (R2 + R3)^*$ it can generate.

Fair Finite Delay Hypothesis for the sequence σ of rule applications:
if a (proper or not) prefix σ_1 of σ leads to a situation where a rule Ri can fire at node n, then
(i) *either* for some subsequences σ_2 and σ_3 of rule applications we have that $\sigma = \sigma_1; \sigma_2; Ri; \sigma_3$ and at the end of σ_2, rule Ri fires at node n (informally, every rule which can fire, actually fires within a finite time),
(ii) *or* there is a (proper or not) prefix of σ for which the Termination Condition holds.

When the algorithm terminates, that is, the Termination Condition holds, the original graph G has been modified into a *directed* spanning tree, say T_d. Then, the undirected spanning tree T of the graph G can be obtained from T_d by considering for every arc $\langle m, n\rangle$ also the symmetric arc $\langle n, m\rangle$.

If we assume that rule $R3$ is applied to every marked node *at most once*, then the sequence of applications of the rules is of the form: $R1\,;(R2 + R3)^{2(|N|-1)}$, where in every prefix the number of $R2$'s is at least that of $R3$'s and, in the whole sequence, the number of $R2$'s is the same of that of $R3$'s, that is, $|N|-1$.

During the computation of the spanning tree T_d the following invariants hold: (i) the marked nodes form a tree with root n_0, and (ii) every node which sent upwards an end-token, is the root of a subtree of the spanning tree T_d.

If after the computation of the spanning tree T, the original graph G should be maintained for further computations (such as the termination detection algorithm of the following Section 4.13), we need to apply the rules of Figure 13 on the preceding page in a conservative manner, in the sense that the deletion of the arcs should not be realized and, instead of deleting arcs, we should simply mark them in some suitable manner.

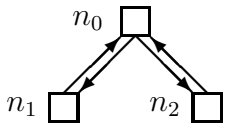
The computation of the directed spanning tree T_d can also be performed by variants of the rules $R1$, $R2$, and $R3$ in which we stipulate that after the application of the rules, *the nodes are left unmarked*. These variants of the rules which we call $R1^{\square}$, $R2^{\square}$, and $R3^{\square}$, respectively, still send and erase mark-tokens and end-tokens and can be depicted as rules $R1$, $R2$, and $R3$ shown in Figure 13, except that every marked node ☒ should be replaced by an unmarked node ☐.

Let us study the termination of the algorithm which uses the rules $R1^{\Box}$, $R2^{\Box}$, and $R3^{\Box}$. For every sequence of rule applications in the set $R1^{\Box}; (R2^{\Box}+R3^{\Box})^*$ we assume the following Finite Delay hypothesis (which is weaker than the Fair Finite Delay hypothesis).

Finite Delay Hypothesis for the sequence σ of rule applications:
if a (proper or not) prefix σ_1 of σ leads to a situation where a rule can fire at a node, then
(i) *either* for some *non-empty* subsequence σ_2 of rule applications we have that $\sigma = \sigma_1; \sigma_2$ (informally, if a rule can fire at a node, then there is a rule, maybe a different one, which actually fires at a node, maybe a different one, within a finite time),
(ii) *or* there is a (proper or not) prefix of σ for which the Termination Condition holds.

If the Finite Delay hypothesis holds then the termination of the algorithm which uses rules $R1^{\Box}$, $R2^{\Box}$, and $R3^{\Box}$, is ensured if the following two extra conditions hold.
Condition $(C1)$: Rule $R1^{\Box}$ is applied to the root node n_0 *only once* and only at the beginning of the computation.
Condition $(C2)$: Rule $R3^{\Box}$ is applied to any given (unmarked) node *at most once*. Actually, in order to ensure termination, it is enough to assume Condition $(C1)$ and Condition $(C2')$: Rule $R3^{\Box}$ is applied to any given unmarked node *at most a finite number of times*.

The following Properties (1) and (2) hold if we apply the original rules $R1$, $R2$, and $R3$, and also if we apply the variants $R1^{\Box}$, $R2^{\Box}$, and $R3^{\Box}$.
Property (1): Rule $R2$ is applied *once* to every non-root node of the graph. For instance, given the following undirected graph with three nodes:



a possible sequence of rule applications for computing a directed spanning tree of the graph with root n_0 is: $R1$; $R2$; $R3$; $R2$; $R3$, where the leftmost applications of rules $R2$ and $R3$ are to the same node (either n_1 or n_2) .
Property (2): Rule $R3$ cannot be applied to a node before a mark-token arrives at it. Indeed, in an undirected, connected graph with at least two nodes and without self-loops, every node as at least one successor node. Thus, if we consider any path from the root of the spanning tree T_d to one of the leaves of T_d, the first application of rule $R3$ to a node in that path is to the leaf of that path.

Our algorithm for computing a spanning tree of a given finite, undirected, connected graph *without self-loops*, can also be used for computing a spanning tree of a finite, undirected, connect graph *with self-loops*, that is, when in the set of edges of the given graph there exists an arc of the form $\langle n, n\rangle$ for some node n. Indeed, if the given graph, call it G_s, has self-loops, it is enough: (i) to consider the graph G which is obtained from G_s by deleting all self-loops, and then (ii) to construct a spanning tree of G according to our rules of Figure 13 on page 52.

Given a graph $G = \langle N, E\rangle$, the three rules depicted in Figure 13 can be realized by three corresponding rewriting rules on 6-tuples, each 6-tuple being an element of $N \times \{\mathit{false}, \mathit{true}\} \times 2^N \times 2^N \times 2^N \times 2^N$. Each 6-tuple encodes a node of G by providing its name in the first component of the 6-tuple and, in the other components, information relative to the marking of the node, the predecessor nodes, the successor nodes, and the mark-tokens and the end-tokens which have been sent to the node.

In particular, if the first component of a 6-tuple is the number n, denoting the node n, then
- the second component is *false* if the node n is unmarked (denoted $\square$) and it is *true* if the node n is marked (denoted $\boxtimes$) (as already mentioned, all nodes are initially unmarked),
- the third component is the set $P(n)$, that is, the set of predecessors of the node n,
- the fourth component is the set $M(n) \subseteq P(n)$ of so called *marking nodes*, that is, the nodes which are predecessors of the node n and have sent a mark-token to the node n and that mark-token has not been erased by an application of rule $R2$ (we have that after the application of rule $R2$ to node n, $M(n) = \{\}$, and we assume that initially, for every $n \in N$, $M(n) = \{\}$),
- the fifth component is the set $S(n)$, that is, the set of successors of the node n, and
- the sixth component is the set $E(n) \subseteq S(n)$ of so called *end nodes*, that is, the nodes which are successors of the node n and have sent an end-token to the node n (we assume that initially, for every $n \in N$, $E(n) = \{\}$).

The rewriting rules, called $\rho 1$, $\rho 2$, and $\rho 3$, corresponding to the rules $R1$, $R2$, and $R3$ of Figure 13 are as follows. (As usual, a rewriting rule of the form: $\ell \Rightarrow r$ means that the left-hand-side ℓ is rewritten into the right-hand-side r, and the other components of the 6-tuple, if any, are left unchanged.)

$\rho 1$: For each node $p \in P(n_0)$, $S(p) \Rightarrow S(p) - \{n_0\}$ (the root has no father)
$\langle n_0, \mathit{false}, P(n_0), M(n_0), S(n_0), \{\}\rangle \Rightarrow \langle n_0, \mathit{true}, \{\}, \{\}, S(n_0), \{\}\rangle$ (marking the root)
For each node $s \in S(n_0)$, $M(s) \Rightarrow \{n_0\}$ (sending $\otimes$ to each successor of the root)

$\rho 2$: Consider a unmarked node $n \neq n_0$ such that there exists a node $i \in M(n)$.
For each node $p \in P(n) - \{i\}$, $S(p) \Rightarrow S(p) - \{n\}$ (node i becomes father of node n)
$\langle n, \mathit{false}, P(n), M(n), S(n), \{\}\rangle \Rightarrow \langle n, \mathit{true}, \{i\}, \{\}, S(n), \{\}\rangle$ (marking node n and erasing all $\otimes$'s arriving at node n)
For each node $s \in S(n)$, $M(s) \Rightarrow M(s) \cup \{n\}$ (sending $\otimes$ to each successor of node n)

$\rho 3$: Consider a marked node n such that $S(n) = E(n)$ (all sons of n have sent $\blacktriangle$ to n) and there exists a node i such that $P(n) = \{i\}$. (node i is the father of node n)
$E(i) \Rightarrow E(i) \cup \{n\}$ (sending $\blacktriangle$ from node n to node i)

In rule $\rho 3$ the condition $M(n) = \{\}$ which appears in Figure 13 on page 52, is not needed (and we did not included) because if the node n is marked then rule $\rho 2$ must have been applied to node n, and in that application all mark-tokens have been erased.

As for the rules $R1$, $R2$, and $R3$, also the rules $\rho 1$, $\rho 2$, and $\rho 3$ are applied in an *atomic* way, in the sense that when one rule is applied to a node, say n, no other can be applied to a node in $P(n) \cup S(n)$.

As for the rules $R1$, $R2$, and $R3$, if we assume that rule $\rho 3$ is applied to every marked node at most once, then the sequence of applications of the rules $\rho 1$, $\rho 2$, and $\rho 3$ is $\rho 1\,;(\rho 2 + \rho 3)^{2(|N|-1)}$, where in every prefix the number of $\rho 2$'s is at least that of $\rho 3$'s and, in the whole sequence, the number of $\rho 2$'s is equal to the number of $\rho 3$'s, that is, $|N|-1$.

After the initial application of the rule $\rho 1$ to the root node n_0, in rule $\rho 2$ we need *not* assume that $n \neq n_0$, because $M(n_0) = \{\}$. When rule $\rho 3$ is applied to a node n, we have that, by construction, $P(n)$ is a singleton. Termination is detected whenever $S(n_0) = E(n_0)$. The first application of rule $\rho 3$ is done to a node which is a leaf of the resulting spanning tree, because initially $E(n) = \{\}$, and we have that $S(n) = E(n)$ iff $S(n) = \{\}$.

The following two properties hold:
(i) when rules $\rho 1$ and $\rho 2$ are applied to an unmarked node n, $E(n) = \{\}$ and does not change its value, and after the application of any of these two rules $M(n) = \{\}$, and
(ii) when rule $\rho 3$ is applied to a marked node n, $M(n) = \{\}$ and does not change its value, and after the application of this rule, the value of $M(n)$ is no longer needed for the application of any of the rewriting rules.

These two properties allow us to implement the distributed algorithm for constructing a spanning tree of a graph by using 5-tuples, instead of 6-tuples. Indeed, we can get rid of the sixth component of the 6-tuples and store the value of $E(n)$ in the fourth component, instead of the sixth one. Then, in order to know the values of $M(n)$ and $E(n)$ from the value of the fourth component, we can use the following two facts whose proof is left to the reader.

(i) If $S(n) = \{\} \ \vee \ P(n) \cap S(n) \neq \{\}$ (that is, node n is a leaf of the spanning tree or rule $\rho 2$ has not been yet applied to node n) then the fourth component stores $M(n)$ and $E(n) = \{\}$.

(ii) If $S(n) \neq \{\} \wedge P(n) \cap S(n) = \{\}$ then the fourth component stores $E(n)$ and $M(n) = \{\}$.

The above two facts are based on the following properties:

(i) *before* the application of rule $\rho 1$ or $\rho 2$ to a node n we have that: $(P(n) \cap S(n) \neq \{\}) \vee S(n) = \{\}$, and

(ii) *after* the application of rule $\rho 1$ or $\rho 2$ to a node n we have that: $(P(n) \cap S(n) = \{\}) \wedge |P(n)| \leq 1$.

We can also get rid of the second component of the 6-tuples. *This amounts to leave all nodes unmarked* and use 4-tuples, instead of 6-tuples or 5-tuples. In this case we use the following rules $\rho 1^{\Box}$, $\rho 2^{\Box}$, and $\rho 3^{\Box}$. (We have added the $\Box$ superscripts to the names of those rules to recall that they operate on nodes which are left unmarked.)

$\rho 1^{\Box}$: For each node $p \in P(n_0)$, $S(p) \ \Rightarrow \ S(p) - \{n_0\}$ (the root has no father)
$\langle n_0, P(n_0), M(n_0), S(n_0)\rangle \ \Rightarrow \ \langle n_0, \{\}, \{\}, S(n_0)\rangle$
For each node $s \in S(n_0)$, $M(s) \ \Rightarrow \ \{n_0\}$ (sending $\otimes$ to each successor of the root)

$\rho 2^{\square}$: Consider a node n such that there exists a node $i \in M(n)$ and (either $S(n) = \{\}$ or $P(n) \cap S(n) \neq \{\}$).
For each node $p \in P(n) - \{i\}$, $S(p) \Rightarrow S(p) - \{n\}$ (node i becomes father of node n)
$\langle n, P(n), M(n), S(n)\rangle \Rightarrow \langle n, \{i\}, \{\}, S(n)\rangle$ (erasing $\otimes$ arriving at node n)
For each node $s \in S(n)$, $M(s) \Rightarrow M(s) \cup \{n\}$ (sending $\otimes$ to each successor of node n)

$\rho 3^{\square}$: Consider a node n such that $S(n) = E(n)$ (all sons of n have sent ▲ to n)
and there exists a node i such that $P(n) = \{i\}$. (node i is the father of node n)
$E(i) \Rightarrow E(i) \cup \{n\}$ (sending ▲ from node n to node i)

It is the case that, having in rule $\rho 2^{\square}$ the extra condition (w.r.t. rule $\rho 2$) 'either $S(n) = \{\}$ or $P(n) \cap S(n) \neq \{\}$', we can apply rule $\rho 2^{\square}$ to every (unmarked) node *at most once*.

In rule $\rho 3^{\square}$ the extra condition $M(n) = \{\}$, which appears in Figure 13 on page 52, is not needed (and we did not included), because: (i) if the given graph has two nodes only, then every sequence of rule applications has initial subsequence: $\rho 1^{\square}$; $\rho 2^{\square}$; $\rho 3^{\square}$ for which the Termination Condition holds, and (ii) if the given graph has more than two nodes, before any firing of the rule $\rho 3^{\square}$ at node n we need $P(n)$ to be a singleton, and in order to have $P(n)$ to be a singleton, we need that rule $\rho 2^{\square}$ has fired at n and this firing made $M(n) = \{\}$.

Note that in rule $\rho 3^{\square}$, instead of referring to the array E, we may equivalently refer to the array M, because we assume that those arrays are stored in the same locations.

When we use the rules $\rho 1^{\square}$, $\rho 2^{\square}$, and $\rho 3^{\square}$, in order to ensure termination we have to assume that Finite Delay hypothesis and the following two conditions.
Condition ($C1$): Rule $\rho 1^{\square}$ is applied to the root node n_0 *only once* and only at the beginning of the computation.
Condition ($C2$): Rule $\rho 3^{\square}$ is applied to every (unmarked) node *at most once*.
This condition on rule $\rho 3^{\square}$ can be enforced by modifying rule $\rho 3^{\square}$ so that the nodes n and i should satisfy the property: $S(n) = E(n) \wedge \exists i,\ P(n) = \{i\} \wedge n \notin E(i)$, instead of the property: $S(n) = E(n) \wedge \exists i,\ P(n) = \{i\}$ (again, referring to the array E or referring to the array M does not make any difference).

Let us make a remark concerning the use of variants of the rules $\rho 1^{\square}$, $\rho 2^{\square}$, and $\rho 3^{\square}$, called $\widetilde{\rho 1}^{\square}$, $\widetilde{\rho 2}^{\square}$, and $\widetilde{\rho 3}^{\square}$, respectively, where we keep the sixth components but we eliminate the second components of the 6-tuples. These variants leave the nodes unmarked. Technically,
(i) we get $\widetilde{\rho 1}^{\square}$ from $\rho 1^{\square}$ by using $\langle n_0, P(n_0), M(n_0), S(n_0), \{\}\rangle \Rightarrow \langle n_0, \{\}, \{\}, S(n_0), \{\}\rangle$, instead of $\langle n_0, P(n_0), M(n_0), S(n_0)\rangle \Rightarrow \langle n_0, \{\}, \{\}, S(n_0)\rangle$,
(ii) we get $\widetilde{\rho 2}^{\square}$ from $\rho 2^{\square}$ by using $\langle n, P(n), M(n), S(n), \{\}\rangle \Rightarrow \langle n, \{i\}, \{\}, S(n), \{\}\rangle$, instead of $\langle n, P(n), M(n), S(n)\rangle \Rightarrow \langle n, \{i\}, \{\}, S(n)\rangle$, and
(iii) we get $\widetilde{\rho 3}^{\square}$ from $\rho 3^{\square}$ by making no changes.

We have that we can compute a spanning tree without applying rule $\widetilde{\rho 2}^{\square}$. This occurs, for instance, for the graph with three nodes depicted on page 54. Indeed, a spanning tree with root n_0 can be computed by the sequence $\widetilde{\rho 1}^{\square}$; $\widetilde{\rho 3}^{\square}$; $\widetilde{\rho 3}^{\square}$ of rule applications.

In the Appendix on page 116 we will present a sequential Java program which implements a distributed algorithm for computing a spanning tree of any finite, undirected,

connected graph which is represented as an *array of nodes*. That algorithm works by leaving each node unmarked. Each node is represented as 3-tuple of arrays as we now explain. We have already seen that a node can be represented as a 4-tuple, instead of a 6-tuple. We can further reduce the 4-tuples to 3-tuples because the first component of every 4-tuple stores the name of the node, and for that name we can simply use the index of the array of nodes that represents the given graph.

4.13 Distributed Termination Detection

We address here the problem of detecting the termination of distributed computations [20]. We are given N processes $P_0, \ldots, P_{N-1}$ which are placed at the nodes of a given *undirected, connected* graph G. We identify each process with the node where it is located. The N processes perform a *main computation* and they exchange *messages* to neighbouring processes in the graph. The arcs of the graph represent communication channels. Each process (and the corresponding node) may be either
- *active*, that is, it still performs a part of the main computation (and in this case the process and the node are labelled by A), or
- *idle*, that is, it has completed the part of the main computation which has been assigned to it by the last message it has received (and in this case the process and the node are labelled by I).
We have that:

(1) Only active processes may send messages.

(2) A process may change from idle to active only on receipt of a message.

(3) A process may change from active to idle at any time (thus, we not assume any knowledge on the duration of the parts of the computations assigned to the processes).

At Point (2) we may replace 'may' by 'must' because of Point (3).

We say that the main computation has *terminated* iff all processes are idle.

We say that termination has been detected by a termination detection algorithm if it is the case that a process, say P_0, enters a predetermined, fixed state whenever the main computation has terminated. To know whether or not the remaining processes have terminated, process P_0 has the ability of sending and receiving messages to all processes in the graph, but this should be done by sending and receiving messages only to and from neighbouring processes. The ability of sending and receiving messages is given to every process in the graph G.

The messages devoted to termination detection are collectively called *signals*. Signals are: (i) either *tokens* or (ii) *repeats*. We will see below how tokens and repeats are used by the termination detection algorithm.

The termination detection algorithm we look for, is an algorithm which: (i) at each node sends or receives tokens or repeats, and (ii) at each node modifies the state of the process at that node.

Moreover, the termination detection algorithm should satisfy the following conditions:

(1) The modification of each process to incorporate termination detection should be independent of the definition of the process.

(2) The termination detection algorithm should not indefinitely delay the main computation.

(3) No new communication channels should be added among the processes.

(4) The termination detection algorithm should operate at each node on the basis of the information, tokens, and repeats available at that node only.

We assume that we have computed a spanning tree T of the given undirected graph G with process P_0 at its root. This can be done by the distributed algorithm we have indicated in the previous Section 4.12 on page 51.

The proposed termination detection algorithm works by making use of waves of tokens and repeats moving along frontiers of the spanning tree T. A *frontier* of a tree is a set F of nodes such that every root-to-leaf path has exactly one node in common with F.

Initially, a contracting *token wave* goes inwards from the leaves to the root, and if it reaches the root without detecting termination, then a new wave, called *repeat wave*, is generated. This new wave moves outwards from the root to the leaves. As soon as this repeat wave reaches the leaves, a new wave of tokens starts moving inwards again. Notice that in some branches of the spanning tree the token wave may be moving inwards while the repeat wave is still moving outwards along other branches.

The algorithm works by applying in a distributed way, as long as possible, the rules that we will indicate in the Figures 14 on the following page and 15 on page 61 below. If the left-hand-side of a rule depicted in these figures holds at a node and the corresponding condition, if any, is true, then the rule fires and the right-hand-side of that rule is realized at that node, regardless of the rules which are applied in other nodes at a previous or later time.

In the spanning tree T of G every node has exactly one parent node, except the root which has no parent. Every node has a (possibly empty) list of children. The list of children is empty iff the node is a leaf.

In Figure 14 and Figure 15 below we have adopted the following conventions. The status s of a node may be either *active* (A) or *idle* (I). The color of a node may be either *white* (□) or *black* (■). A token t may be either *white* (△) or *black* (▲). A repeat is denoted by •. A pair of symmetric arcs between any two nodes is depicted as a single arc without arrowheads.

Here are the two *rules* $M1$ *and* $M2$ *for the main computation* (see also Figure 14 on the following page).

Rule $M1$. It is for sending a message along an arc. An active process sends a message m to one of its neighbours, and assigns to it a part of the main computation which is encoded by that message. The process which sends the message m becomes black, and this color encodes the fact that the process which *receives* the message m, has begun and not yet completed the part of the main computation encoded by m.

Rule $M2$. At any time an active process may become idle regardless of its color. It becomes idle when it has completed the part of the main computation encoded by the last message it has received.

We do *not* assume that nodes remain active for a finite time only. Thus, if a node remains active for an infinite time, then rule $R2$ never fires at that node and the termination detection algorithm should never detect termination.

Here are the five *rules for the termination detection algorithm* (see also Figure 15 on page 61).

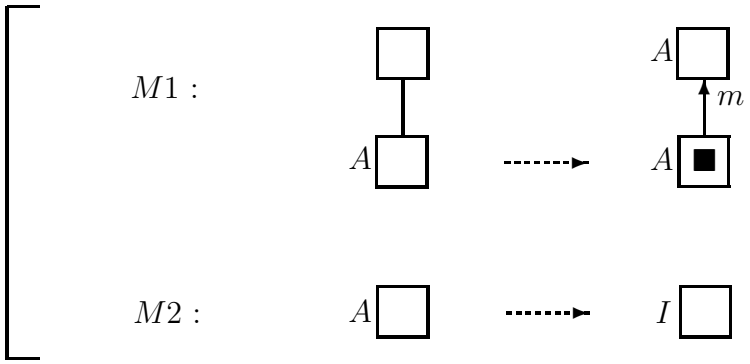


Fig. 14. Rules of the main computation of the termination detection algorithm. The status s of a node may be either *active* (A) or *idle* (I). ■ denotes that the color of the node is *black*. m is a message of the main computation.

Rule $T1.down$ is for the root of the spanning tree. When the root P_0 has received a token from each of its children, then P_0 destroys those tokens, becomes white, and sends a repeat to each of its children iff either (i) P_0 is active, or (ii) P_0 is black, or (iii) P_0 has received a black token.

Rule $T2.up$ is for any internal node of the spanning tree (neither the root nor the leaves). When an internal node P_i has received a token from each of its children and P_i is idle, then P_i destroys those tokens and sends a new token t to its parent. The token t is black iff either P_i is black or any of the tokens received from the children is black, otherwise the token is white.

Rule $T2.down$ is for any internal node of the spanning tree (neither the root nor the leaves). When the internal node has received a repeat, it destroys that repeat and sends a repeat to each of its children.

Rule $T3.up$ is for any leaf. A leaf sends its white token to its parent iff the leaf is idle.

Rule $T3.down$ is for any leaf. A leaf which receives a repeat, destroys it, and acquires a white token.

Initially:
- all nodes are white;
- at least one node is active (and thus, it may become black);
- each leaf has a white token and every other node has neither tokens nor repeats.

These are *invariants* of the algorithm:
- an active node does *not* send any token (see Figure 15) and thus, it is impossible for the termination detection algorithm to indefinitely delay the main computation;
- when a node sends a token it is left without tokens, and analogously, when it sends a repeat it is left without repeats;
- tokens at the leaves are always white.

We will show below that termination is detected by the process P_0 at the root when at a given time $\bar{t}$:

(i) the process P_0 at the root is idle, and

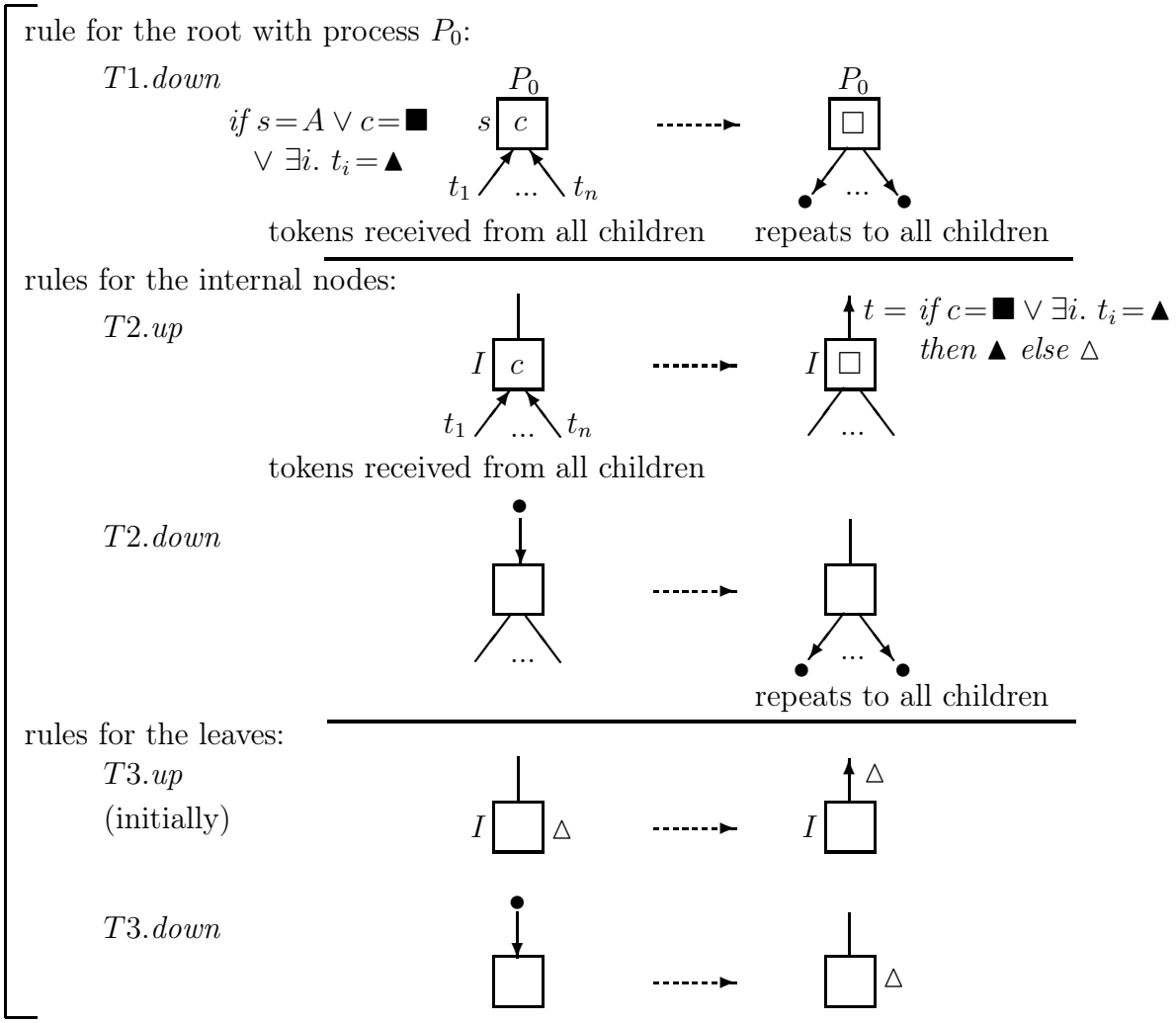


Fig. 15. Rules of the termination detection algorithm. The status s of a node may be either *active* (A) or *idle* (I). The color c of a node may be either *white* (□) or *black* (■). A token may be either *white* (△) or *black* (▲). • is a repeat.

(ii) the color of the root is white, and

(iii) there is a time t before $\bar{t}$ such that in the interval $[t, \bar{t}]$ every child of the root has sent to the root one token only and that token is white (it is △).

If Conditions (i), (ii), and (iii) above hold, then the rule $T1.down$ does not fire and no new repeat wave is generated. At this point P_0 has detected termination and it may tell all other processes to halt. We do not describe here how this last communication may be realized by suitable messages sent along the arcs of the spanning tree.

The choice between the firing of the rules of the main computation and those of the termination detection algorithm is *nondeterministic*. Thus, the main computation need not be delayed by the termination detection algorithm.

The algorithm is correct independently of the fact that processes have buffers to store the incoming messages and signals, provided the delay between sending and receiving

a message or a signal is sufficiently small, that is, the propagation along the arcs of the spanning tree is sufficiently fast. Thus, correctness holds provided that the following hypothesis holds.

> *Fair Finite Delay Hypothesis for rules and messages*: (i) every rule which can fire, actually fires within a finite time, and (ii) every message takes a finite time to reach destination.

The proof of correctness of the termination detection algorithm is as follows [20]. We first show partial correctness. We need the following definition.

Definition 5. Given an undirected, connected graph G and a spanning tree T of G, a node n of G is said to be *outside a set* A of nodes of G iff $n \notin A$ and the path from the root of T to n (including the root) contains an element of A.

In this definition it is irrelevant whether or not the node n is included in the path from the root of T to n.

Let S be the set of nodes of the spanning tree T with one or more tokens, regardless of the colors of the tokens. Let us consider the invariant Inv defined as follows:

$$\begin{aligned} Inv \equiv\ & (\text{all nodes outside } S \text{ are idle} \\ & \vee \text{ some nodes not outside } S \text{ is black} \\ & \vee \text{ some node in } S \text{ has a black token}) \end{aligned}$$

If we take S to be the set of all leaves, we have that the invariant Inv is initially true because no node is outside S. The invariant Inv is maintained true for each firing of the rules of Figures 14 and 15 by assuming that the set S is modified by the rules $T2.up$ and $T3.up$ only, and it is modified as follows:

the node n which sends the token is removed from S, and the parent p of that node is added to S if it is not already an element of S, that is, $S := (S \cup \{p\}) - \{n\}$.

Now, since $S = \{\text{root}\}$ implies that all nodes but the root, are outside S, we have that:

$(Inv \wedge S = \{\text{root}\} \wedge$ the root is white and idle $\wedge$ all tokens at the root are white)
$\Rightarrow$ all nodes are idle (that is, the main computation has terminated)

Moreover, since Inv is preserved during the firing of the rules of Figures 14 and 15, the main computation has terminated if $S = \{\text{root}\} \wedge$ the root is white and idle $\wedge$ all tokens at the root are white.

The termination detection algorithm terminates because:
(i) to test the firing conditions of each rule of Figures 14 and 15 takes a finite amount of time, and
(ii) if all processes are idle then a finite number of firings of the rules of Figure 15 is sufficient to allow process P_0 to detect termination.

As indicated in [20], we have that the complexity of the termination detection algorithm is $O(N \times m)$, where N is the number of nodes in the graph G (that is, the number of processes) and m is the number of messages generated by the main computation according to rule $M1$.

4.14 Lock-free and Wait-free Synchronization

The synchronization techniques based on semaphores, critical regions, conditional critical regions, and monitors, which we have described in the preceding sections, are *not* suitable for fault-tolerant and real-time systems. Indeed, if a process which has mutually exclusive access to a resource, fails, then no process can access that resource any more. Likewise, in real-time systems it may be required that a process executes its critical section before a deadline expires. This may not be guaranteed if there is no a priori bound for the time in which a process is in its critical section.

Synchronization policies which may be used in fault-tolerant and real-time system cannot rely on semaphores, critical regions, conditional critical regions, or monitors. We should use new synchronization techniques which are called *lock-free* techniques. We will not enter into this topic here and the reader may refer to [10].

If we want to ensure that the execution of a critical section is performed within a bounded number of steps we need more sophisticated synchronization techniques, called *wait-free* techniques. Also the study of these techniques is left to the enthusiastic reader who may refer, for instance, to [10].

5 Concurrent Computations in Java

In this section we will look at Java programs which implement some of the mechanisms and techniques we have introduced in the previous sections. This material is derived from [10].

Let us begin by stating the difference between a *process* and a *thread*. A process has three parts: (i) the *code* (that is, the sequence of machine instructions to be executed), (ii) the *data* (that is, the memory locations for storing global variables), and (iii) the *stack* (that is, the memory locations for storing local variables and activations records of the procedure calls).

Processes which share code and data and have distinct stacks are said to be *lightweight processes*, or *thread*s. Java allows us to *construct* and *run* threads. In Java there exists a predefined class Thread that we can *extend* to construct threads. The actual construction of a thread occurs when a **new** instruction is executed (see line 1 in the program below). We *override* the method run() of the class Thread to define the behaviour of any constructed thread (see lines 2 and 3). The actual running of a thread begins only after the execution of the *start*() method relative to the thread (see line 4).

The reader should notice that sometimes for simplicity reasons, we will use the word 'process' as a synonymous of 'thread'. Notice also that according to the Java terminology, people often say that threads are 'created', rather than 'constructed'.

The following is an example of a Java program which constructs a thread and makes it run.

```
public class MerryChristmas extends Thread {          // 1
  public void run() {                                 // 2
    System.out.println("Merry Christmas!");           // 3
  }
  public static void main(String [] args) {
  MerryChristmas t = new MerryChristmas();
  t.start();                                          // 4
  }
}
```

Let us save this class in a file named MerryChristmas.java. Then, if we execute the commands:

```
javac MerryChristmas.java
java  MerryChristmas
```

we get:

```
Merry Christmas!
```

In Java there is an alternative way to construct threads. We illustrate this alternative way by presenting a concrete example of how to construct from a single class several distinct objects which may run as distinct threads.

Let us consider the following class Counter saved in a file named RunnableCounter.java together with the class RunnableCounter:

```
class Counter {
  private int value;
  public Counter (int value) {
    this.value = value;
  }
  public void setValue (int value) {
    this.value = value;
  }
  public int getValue () {
    return this.value;
  }
  public int addOne () {
    this.value ++;
  }
}
```

Now let us construct distinct objects from this class Counter. We want these objects to run as distinct threads. In order to do so we should construct a new class which extends both the class Counter and the class Thread (which is a class provided by the standard Java environment). We face a problem here because Java does not allow *multiple inheritance*, that is, it does *not* allow a concrete class to inherit from more than one class.

Note 3. In Java the only form of multiple inheritance is the one of a class which inherits from a set of interfaces. Indeed, the declaration:

class C **implements** I1, ..., In {*classbody*}

realizes the multiple inheritance of the concrete class C from the interfaces I1, ..., In. □

This problem of extending both the class Counter and the class Thread, is solved by introducing the following class RunnableCounter which **extends** Counter and **implements** Runnable, where Runnable is a standard Java interface which declares a single method: **public void** run().

The declaration of the class RunCounter is as follows. We assume that it is saved, together with the above class Counter, in a single file named RunnableCounter.java.

```
public class RunnableCounter extends Counter implements Runnable {
  public RunnableCounter (int v) {
    super(v);
  }
  public void run() {
    for (int i = 0; i < 3; i++) {System.out.print(" " + getValue()); addOne();}
  }
```

```
    public static void main (String[] args) {
      RunnableCounter c1 = new RunnableCounter(10);
      Thread t1 = new Thread(c1);
      t1.start();
      RunnableCounter c2 = new RunnableCounter(c1.getValue() + 10);
      Thread t2 = new Thread(c2);
      t2.start();
    }
}
```

Then, if we execute the commands:

```
javac RunnableCounter.java
java  RunnableCounter
```

we get:

```
10 11 12 20 21 22
```

Actually, the output we get depends on the Java runtime system in use. One may also get:

```
10 11 12 23 24 25
```

Nothing can be said about the progress of the thread $t1$ when the thread $t2$ starts running. In particular, the thread $t1$ may not have finished its activities when thread $t2$ starts running.

In Java we can make a thread to wait for the termination of another thread by using *join*(). The *join*() operation requires a **try-catch** statement. The following class FibonacciThread which computes the Fibonacci numbers, illustrates this mechanism. We assume that this class is saved in a file named FibonacciThread.java.

```
public class FibonacciThread extends Thread {
   int n;
   int value;
   public FibonacciThread(int n){
   this.n = n;
   }
   public void run(){
   if (n == 0) value = 0;
   else if (n == 1) value = 1;
   else {FibonacciThread fib1 = new FibonacciThread(n−1);
         FibonacciThread fib2 = new FibonacciThread(n−2);
         fib1.start(); fib2.start();
         try {fib1.join(); fib2.join();
         } catch (InterruptedException e){};
         value = fib1.value + fib2.value;
        }
   }
```

```
    public static void main(String [] args){
      int n = Integer.parseInt(args[0]);
      FibonacciThread fib = new FibonacciThread(n);
      fib.start();
      try {fib.join();
      } catch (InterruptedException e){};
      System.out.println("fib("+ n +") = " + fib.value);
    }
  }
```

Then, if we execute the commands:

```
javac FibonacciThread.java
java  FibonacciThread  8
```

we get:

fib(8) = 21.

Threads can communicate with each other by writing and reading static fields, non-static fields, and array elements. Threads cannot communicate by using local variables and method parameters.

5.1 Mutual Exclusion in Java

In this section we consider the problem of mutual exclusion between two threads: T_1 and T_2. In order to realize the mutually exclusive access to a critical section by one thread at a time, we implement the following interface which we call EntryExitProtocol.

```
public interface EntryExitProtocol {
  public void entryProtocol(int processId);
  public void exitProtocol(int processId);
}
```

We assume that this interface is stored in a file named EntryExitProtocol.java.

The EntryExitProtocol interface has two methods: a first one, called entryProtocol(i), which is executed by thread T_i, for $i = 1$ or 2, before entering its critical section, and the second one, called exitProtocol(i), which is executed by thread T_i, for $i = 1$ or 2, after exiting its critical section. As we will see, the entryProtocol(i) and exitProtocol(i) methods correspond to the operations of acquiring and releasing a lock as explained in Section 7.6.

Peterson's algorithm implements the EntryExitProtocol interface as indicated by the following class, named PetersonTwoProcesses, saved in a file named PetersonTwoProcesses.java. In this implementation we have assumed that the integer 1 identifies thread T_1 and, likewise, the integer 2 identifies thread T_2 (this is in accordance with our conventions of Section 4.11).

```
public class PetersonTwoProcesses implements EntryExitProtocol {
   boolean q1 = false;
   boolean q2 = false;
   int s = 1;
   public void entryProtocol(int i){
      if (i == 1)   { q1 = true;  s = 1;  while (q2 && s != 2); }
      else          { q2 = true;  s = 2;  while (q1 && s != 1); }
   }
   public void exitProtocol(int i) {
      if (i == 1)   { q1 = false; }
      else          { q2 = false; }
   }
}
```

Notice that the class PetersonTwoProcesses works correctly in the sense it ensures that the two threads T_1 and T_2 access their critical sections in a mutually exclusive way, only if the two threads are indeed identified by the integers 1 and 2, respectively. To assume this identification of the two threads means that they have to share some extra global information besides the information which is already shared between them by the fact that they both access the global variables $q1$, $q2$, and s.

We stipulate that the PetersonTwoProcesses class is stored in a file named PetersonTwoProcesses.java. In order to see the PetersonTwoProcesses class in action, we may use the following two classes ProtocolThread and PetersonProtocolTester both stored in a single file named PetersonProtocolTester.java:

```
class ProtocolThread extends Thread {
   int id;
   private static EntryExitProtocol protocol;
   public ProtocolThread (int id, EntryExitProtocol protocol) {
      this.id = id;
      this.protocol = protocol;
   }
   private void sleeping(int id){
      try { if (id == 1) { sleep(50); } else { sleep(300); }          // (†)
      } catch (InterruptedException e) {};
   }
   void nonCriticalSection(int id) {
      System.out.print(" " + id);
      sleeping(id);
   }
   void CriticalSection(int id){
      System.out.print(" [" + id);
      sleeping(id);
      System.out.print("]");
   }
```

```
    public void run() {
      while (true) {
        nonCriticalSection(id);
        protocol.entryProtocol(id);
        CriticalSection(id);
        protocol.exitProtocol(id);
      }
    }
}
public class PetersonProtocolTester {
    public static void main(String [] args) {
                                    // testing Peterson's algorithm for 2 processes
      EntryExitProtocol petersonTwoProcesses = new PetersonTwoProcesses();
      new ProtocolThread(1, petersonTwoProcesses).start();
      new ProtocolThread(2, petersonTwoProcesses).start();
      }
}
```

We assume that the file PetersonProtocolTester.java is placed in a single folder together with the files: (i) EntryExitProtocol.java, and (ii) PetersonTwoProcesses.java. By executing the commands:

```
javac PetersonProtocolTester.java
java  PetersonProtocolTester
```

we get:

1 2 [1] 1 [1] 1 [1] 1 [2] 2 [1] 1 [1] 1 [1] 1 [2] 2 [1] 1 [1] 1 [1] 1 [2] 2 ...

In this output the numbers 1 and 2 denote the thread T_1 and T_2, respectively. The beginning of each critical section is denoted by '[', and the end of each critical section is denoted by ']'. A number without enclosing square brackets denotes that the corresponding thread is inside the non-critical section. A number within enclosing square brackets denotes that the corresponding thread is inside the critical section. Thus, the above output shows that the two threads T_1 and T_2 are not at the same time inside their critical section.

The command *sleep*(n), provided by the Java class Thread, is used for delaying for n milliseconds the thread which executes it.

In the output we have given above, the reader can verify that while *process*2 is in its non-critical section (and it stays there for 300 ms as indicated in the statement (†) of the sleeping(id) method), *process*1 which takes 100 ms to cycle through its non-critical and critical sections, can get into its critical section at most three times.

The reader should also notice that in the above Java implementation of Peterson's algorithm, we get mutual exclusion when accessing critical sections *without* using Java **synchronized** methods. Indeed, the goal of Peterson's algorithm is exactly that of achieving mutual exclusion *at the level of a sequence of commands*, i.e., the sequence of commands of the critical section, starting from mutual exclusion guaranteed by the hardware *at the level of a single assignment*, i.e., the assignment of the shared variables $q1$, $q2$, and s. The mutual exclusion at the level of the single assignment means that, for instance, the

variable s is assumed to have, at each instant in time, a value which is either 1 or 2, even if the method entryProtocol(id) is concurrently executed by two threads.

We leave to the reader the proof that the given implementation of Peterson's algorithm guarantees mutual exclusion, absence of deadlock, and bounded overtaking.

5.2 Monitors in Java

In this section we will present a Java implementation of the monitor primitive proposed by [13].

In Java a monitor is an object whose fields are all private and they are manipulated only via methods which are executed in a mutually exclusive way in the sense that for each monitor, at any time, there exists at most one thread which executes a method of that monitor. Thus, in particular, if a monitor has two or more methods, then at any time at most one of them can be executed, even if the running threads are two or more.

In Java mutually exclusive execution of the methods of a monitor is obtained by writing the keyword **synchronized** in every method of the monitor (see the examples presented in the following sections). We say that every method of a monitor is *synchronized*.

In general, in Java an operation on an object (or array, or class) is guaranteed to be performed in a mutually exclusive way by the use of a *lock* associated with that object (or array, or class). We require that for any object (or array, or class): (i) the lock can be held by at most one thread at a time, (ii) the lock must be obtained from the system before starting the operation on that object (or array, or class), and (iii) the lock must be returned to the system after ending the operation on that object (or array, or class).

In order to guarantee that an operation on an object (or array, or class) is performed in a mutually exclusive way, it is necessary that every thread before starting an operation on that object (or array, or class), gets the associated lock and, for methods of monitors, this is enforced by the presence of the keyword **synchronized**.

Now let us examine how a thread, say u, which executes a method of a monitor object, say o, uses the lock on o. (That lock on the monitor o was given to u before it started the execution of the method.) In particular, let us assume that the thread u while executing the method, requires a resource to become available and this fact is encoded by a boolean variable which has to become true. The thread u starts waiting for the availability of that resource by releasing the lock on the object o and this is done by executing $o.wait()$. At that point the thread u enters the 'Waiting for o' state, and it is added to a queue associated with that state (see Figure 16 below). In that queue, which we call the *condition queue*, the thread u waits for a notification on the object o. Such notification has to come from another thread which has obtained the lock on the object o and has performed either an $o.notify()$ or an $o.notifyAll()$ operation. The notifying thread does not lose the lock on o, and after notification, the thread u must get the lock on o before it can proceed.

Thus, there are two cases: either (i) a thread performs an $o.notify()$ operation, or (ii) a thread performs an $o.notifyAll()$ operation. In Case (i) a thread which is in the condition queue of the monitor o, is removed from that queue, goes into the 'Locking o' state, and is inserted into a different queue, which we call *mutex queue*, associated with the 'Locking o' state. In Case (ii) all threads which are in the condition queue of the

monitor o, are removed from that queue, go into the 'Locking o' state, and are inserted into the mutex queue (see Figure 16).

A thread which is in the 'Locking o' state, can get the lock for the object o from the system and then it will be made *enabled* to run, that is, it can enter the 'Enabled' state. Then the scheduler will choose an enabled thread and will make it run on the processor, according to a given processor allocation policy (see Figure 16).

Thus, in Java every monitor object o has two queues: the *mutex queue* and the *condition queue.*

(1) The *mutex queue* holds every thread which waits for the execution of a method of the monitor o. This waiting is due to the fact that monitor methods are synchronized and they can only be executed in a mutually exclusive way.

(2) The *condition queue* holds every thread which waits for a condition to become true and that condition depends on a field of the monitor o.

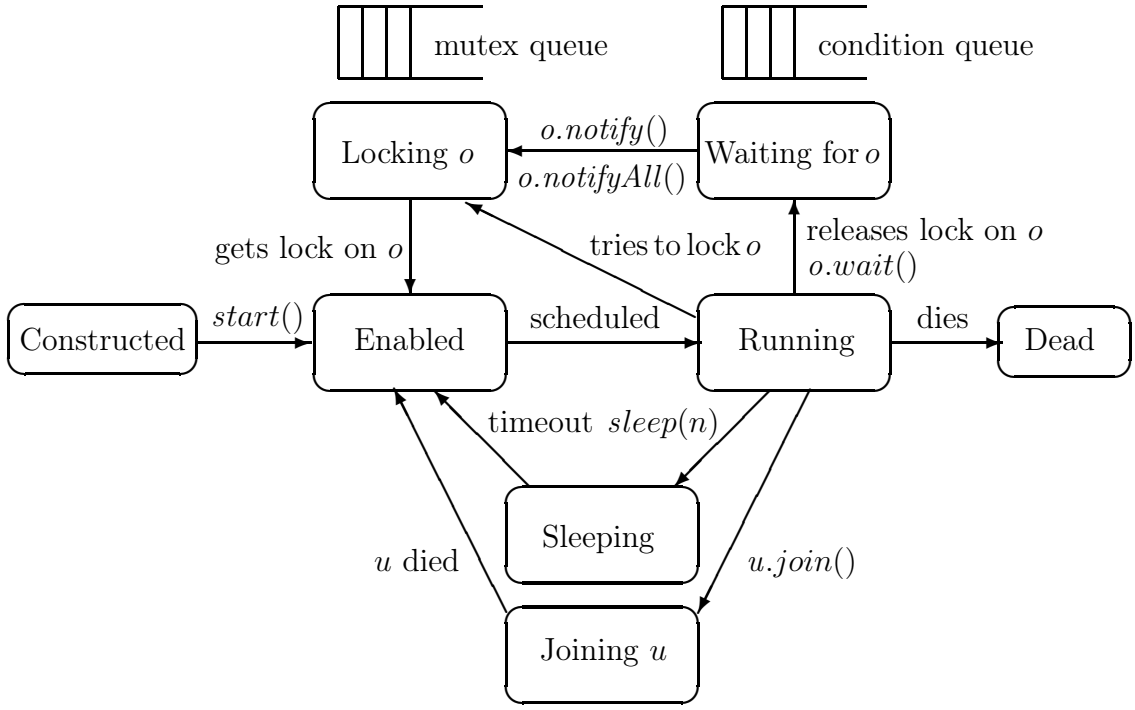


Fig. 16. Simplified diagram of states and transitions for a Java thread u relative to an object o. The commands $o.wait()$ and $sleep(n)$ are performed by the same thread u. The commands $start()$, $o.notify()$, $o.notifyAll()$, and $u.join$ are performed by other threads different from u. Other actions referring to the thread u are: 'scheduled', 'dies', 'tries to lock o', 'timeout', 'gets lock on o', and 'releases lock on o'. A thread different from u, after performing $u.join()$, waits for u to die and when this happens, it becomes enabled.

Figure 16 shows the states and the transitions of a Java thread u relative to an object o. (Actually, for simplicity reasons, in that figure we have depicted only those transitions

which are necessary for understanding the examples we have presented below.) The complete diagram with all the transitions can be found in [19]). The state 'Locking o' is a collection of states, one for each object o. Analogously, 'Waiting for o' is a collection of states, one for each object o, and 'Joining u' is a collection of states, one for each thread u. For further details the reader may refer to [11,19].

Notice that it is the thread which performed an $o.notify()$ (or $o.notifyAll()$) operation that continues its execution on the processor, not the thread which was notified. This is *not* what happens in the monitor proposal by Prof. Hoare [13]. In Hoare's proposal, in fact, a thread which is removed from the condition queue, starts immediately running without the intervention of any other thread. Hoare's proposal has the advantage that the thread which is removed from the condition queue and starts running, it can do so without checking again the value of any variable. The fact that the conditions for running are satisfied is ensured by the notifying thread (i.e., the signalling process in Hoare's terminology).

On the contrary, in a Java monitor the thread which was waiting on a condition queue and is then enabled to run, before it can actually run, has to check the value of the condition variable associated with that queue. This check is required because in the meantime the condition variable might have become false again. Thus, we have that a Java thread that: (i) runs a (synchronized) method of a monitor o, and (ii) waits for a condition B to become true when B depends on a (private) field of the monitor o, should execute the following typical program fragment:

while (!B)
  try { $o.wait()$;
  } **catch** (InterruptedException e) {};

(see, for instance, Sections 5.4 and 5.5). Notice that in Java an $o.wait()$ operation requires a **try-catch** construct, and recall that all methods of a Java monitor are synchronized and all fields are private.

The reader should notice also that in a Java monitor o there is a *unique* condition queue, common to all conditions relative to that monitor, while in the monitor proposal by Prof. Hoare [13] for any monitor there exists a distinct condition queue for each condition variable.

Finally, the reader should notice an important difference between a $wait(s)$ operation performed on a semaphore s and a *wait* operation performed by a process while executing a procedure of a Hoare's monitor. We have that a process which performs a $wait(s)$ operation on a semaphore s, continues running for checking the value of s (see Section 4.3) and there is the busy waiting phenomenon. On the contrary, in a Hoare's monitor a process after performing a *cond.wait* operation on a variable *cond* of type *condition,* stops running and it can be resumed only when a *cond.signal* operation is performed. Thus, the busy waiting phenomenon does not occur.

Similarly to the case of a Hoare's monitor, the busy waiting phenomenon is not present in a Java monitor. Indeed, after a Java thread has performed an $o.wait()$ operation on a monitor o (because, for instance, that thread has to wait for a condition variable to become true), that thread releases the lock on o, stops running, and enters the condition queue of the 'Waiting for o' threads (see Figure 16). However, as already mentioned, in a Java

monitor, contrary to the case of a Hoare's monitor, when a Java thread starts running again after it has performed an *o.wait*() operation, it has to check, before any other operation, that the value of the condition variable which forced that *o.wait*() operation, is indeed the expected one. Otherwise, that thread has to perform an *o.wait*() operation again.

5.3 Bounded Buffer Monitor in Java

The following class BoundedBufferMonitor realizes a Java monitor for a bounded buffer, called *bBuffer*, of size $N\,(> 0)$. The size of the buffer is the number of cells available in the buffer. We assume that each cell contains an object of the Java class Object.

We have a variable *count* which stores the number of items present in the buffer. Thus, $0 \leq count \leq N$.

The array index *in*, with $0 \leq in \leq N-1$, identifies to the cell of the buffer where an item should be put in by the put(*item*) method.

The array index, call it *out*, with $0 \leq out \leq N-1$, which identifies the cell of the buffer where an item should be taken from by the get() method, can be computed from the value of *in* and the value of *count* by using the Java remainder operator %, as we now indicate.

Recall that for any integer k and any integer $N \geq 2$,

$$\begin{array}{lll} k\,\%\,N & =_{def} & \textbf{if } k < 0 \textbf{ then } -\,remainder((-k)/N) \\ & & \textbf{else if } k = 0 \textbf{ then } 0 \\ & & \textbf{else } remainder(k/N) \end{array}$$

Thus, for any $k \geq 0$ and any integer $N \geq 2$, we have that: $k\,\%\,N = k\,mod\,N$. Hence,

$out\ =\ (in - count)\ mod\ N\ =\ (in - count + N)\ mod\ N\ =\ (in - count + N)\,\%\,N$

(see statement (†) in the method get() below).

The two statements below with the comment '// tracing' are needed for tracing the behaviour of the buffer and they can be deleted if so desired. (In Section 4.8 the reader may find a similar monitor realized according to the approach of Prof. Hoare. In that monitor, for historical reasons, we used the identifier *last*, instead of the identifier *in*.)

Let us assume that the BoundedBufferMonitor class is saved in a single file named BoundedBufferMonitor.java.

```
public class BoundedBufferMonitor {
   private Object [] bBuffer;
   private final int N;
   private int in = 0;
   private int count = 0;

   public BoundedBufferMonitor(int n) {
   this.bBuffer = new Object [n];
   this.N = n;
```

```
    public synchronized void put(Object item) {
       while (count == N )
           try { wait();
           } catch (InterruptedException e){};
       bBuffer[in] = item;
       System.out.println("-> " + item.toString());          // tracing
       in = (in + 1) % N;
       count ++;
       notify();
    }
```

```
    public synchronized Object get() {
       while (count == 0)
           try { wait();
           } catch (InterruptedException e){};
       Object item = bBuffer[(in - count + N) % N];          // (†)
       System.out.println("    " + item.toString() + " ->");  // tracing
       count--;
       notify();
       return item;
    }
}
```

Notice that in the above put(*item*) method we cannot erase the keyword **synchronized**. Indeed, let us consider the case where two threads concurrently try to put a new value in a buffer where there is only one cell available, say the one in position $N-1$. Without the keyword **synchronized**, it may happen that the result of the concurrent running of these two threads together determine the overwriting of the cell in position 0, because 0 is the new value of the variable *in* after the execution of one of the two threads.

Analogously, in the get() method we cannot erase the keyword **synchronized**.

The BoundedBufferMonitor class can be viewed in action by using the following three classes all saved in a single file named BoundedBufferMonitorTester.java:

(i) the Producer class,

(ii) the Consumer class, and

(iii) the BoundedBufferMonitorTester class.

For the time being do not consider the comments '// — (M)' which we will explain later.

We also assume that the BoundedBufferMonitorTester.java file is in the same folder where we have saved the BoundedBufferMonitor.java file.

```
class Producer extends Thread {
  private BoundedBufferMonitor bBuffer = null;                    // — (M)
  public Producer(BoundedBufferMonitor bBuffer) {                 // — (M)
    this.bBuffer = bBuffer;
  }
  public void run() {
    Object item = new Integer(0);
    while (true) {
      item = new Integer(((Integer)item).intValue()+1);           // " item ++;"
      bBuffer.put(item);
      try { sleep(10);                                            // (†)
      } catch (InterruptedException e) {}
    }
  }
}
```

```
class Consumer extends Thread {
  private BoundedBufferMonitor bBuffer = null;                    // — (M)
  public Consumer(BoundedBufferMonitor bBuffer) {                 // — (M)
    this.bBuffer = bBuffer;
  }
  public void run() {
    Object item;
    while (true) {
      item = bBuffer.get();
      try { sleep(100);                                           // (†)
      } catch (InterruptedException e) {}
    }
  }
}
```

```
public class BoundedBufferMonitorTester {                         // — (M)
  public static void main(String[] args) {
    final int N = Integer.parseInt(args[0]);
    BoundedBufferMonitor bBuffer = new BoundedBufferMonitor(N); // — (M)
    new Producer(bBuffer).start();
    new Consumer(bBuffer).start();
  }
}
```

Then, if we execute the commands:

```
javac BoundedBufferMonitorTester.java
java  BoundedBufferMonitorTester 3
```

we construct a bounded buffer *bBuffer* with size $N = 3$ and we get the following output:

```
-> 1
   1 ->
```

```
−> 2
−> 3
−> 4
     2 −>
−> 5
     3 −>
−> 6
     ...
```

Obviously, this output behaviour depends on the *sleep* time we have indicated in the Producer and Consumer classes above (see the two lines marked with (†)). For a buffer of size $N = 1$ we get:

```
−> 1
     1 −>
−> 2
     2 −>
−> 3
     3 −>
     ...
```

5.4 Counting Semaphore in Java

The Java implementation of a *counting semaphore*, that is, a semaphore which can take any integer value (either negative or null or positive, depending on its initial value and the code of the programs which use it), is given by the following CountingSemaphore class. We assume that this class is stored in a file named CountingSemaphore.java.

```
public class CountingSemaphore {
   private int value;
   public CountingSemaphore(int value){  // value is the initial value of
      this.value = value;                // the CountingSemaphore
   }
   public synchronized void Wait() { // corresponds to wait(s) for
      while (value <= 0)             // a CountingSemaphore s (see Section 4.3)
         try { wait();
         } catch (InterruptedException e){ };
      value−−;
   }
   public synchronized void Signal(){ // corresponds to signal(s) for
      value ++;                       // a CountingSemaphore s (see Section 4.3)
      notify();
   }
}
```

Notice that the above implementation of a counting semaphore is the direct Java translation of the original Dijkstra's proposal presented in Section 4.3. We will use counting semaphores in Section 5.6 below.

5.5 Binary Semaphore in Java

The Java implementation of a *binary semaphore* is given by the following BinarySemaphore class. We assume that this class is stored in a file named BinarySemaphore.java.

```
public class BinarySemaphore {
   private boolean value;
   public BinarySemaphore(boolean value){
      this.value = value;
   }
   public synchronized void Wait() {    // corresponds to wait(s) for
      while (value == false)            // a BinarySemaphore s (see Section 4.3)
         try { wait();          // the process joins the queue of waiting processes
         } catch (InterruptedException e){ };
      value = false;
   }
   public synchronized void Signal(){   // corresponds to signal(s) for
      value = true ;                    // a BinarySemaphore s (see Section 4.3)
      notify();
   }
}
```

This implementation of a binary semaphore is the direct Java translation of the binary semaphore presented in Section 4.3. In order to get mutual exclusion on a critical section, say *CS,* by using a binary semaphore, say *mutex*, first we must declare the variable *mutex* by using, for instance, the following declaration:

```
BinarySemaphore mutex = new BinarySemaphore(true);
```

and then we must perform on the variable *mutex*, before and after the critical section *CS*, a *Wait*() and a *Signal*() operation, respectively, as follows:

```
mutex.Wait();   // wait if another thread is in its critical section
CS;             // critical section
mutex.Signal(); // notify other threads waiting for executing their critical section
```

We will use a binary semaphore for ensuring mutual exclusion in the example presented in the following Section 5.6.

According to our implementation, a binary semaphore can be viewed a particular instance of a counting semaphore by encoding the value 1 by *true* and the value 0 by *false.*

5.6 Bounded Buffer with Binary and Counting Semaphores in Java

Now we present the Java implementation of a bounded buffer, called *bBuffer*, of size $N\,(>0)$ which, instead of a monitor, uses binary and counting semaphores. This implementation consists of the following class BoundedBuffer. The statements for tracing can be deleted if not necessary. The array indexes *in* and *out* point to the cells of the buffer

where items should be put in and taken from, respectively. We stipulate that the indexes *in* and *out* range over the set $\{0, \ldots, N\}$ and they are both initialized to 0.

Notice that the number of the items present in the buffer is *not* determined by the values of *in* and *out*. Indeed, if *in*=*out* then the number of items can be either 0 or N.

The two statements below with the comment '// tracing' are needed for tracing the behaviour of the buffer and they can be deleted if so desired.

The reader may also refer to Section 4.5 where we have presented the same bounded buffer example written in an abstract programming language. Notice that in our Java implementation below, we use the two semaphores *mutex_P* and *mutex_G*, instead of a single binary semaphore, and thus, it is possible to execute the method put(*item*) together with the method get(), at the same time, by two distinct threads on the same object of the BoundedBuffer class. However, those two methods will never be executed at the same time on the same cell, because the counting semaphores *notEmpty* and *notFull* will force one of the two methods to wait.

We assume that the BoundedBuffer class is saved on a file named BoundedBuffer.java.

```
public class BoundedBuffer {
   private Object [] bBuffer;
   private final int N;
   private int in = 0;
   private int out = 0;
   BinarySemaphore mutexP = null;
   BinarySemaphore mutexG = null;
   CountingSemaphore notEmpty = null;
   CountingSemaphore notFull = null;

   public BoundedBuffer(int N) {
   this.bBuffer = new Object [N];
   this.N = N;
   this.mutexP = new BinarySemaphore(true);
   this.mutexG = new BinarySemaphore(true);
   this.notEmpty = new CountingSemaphore(0);
   this.notFull = new CountingSemaphore(n);

   public void put(Object item) {
      notFull.Wait();        // wait until the buffer becomes not full
      mutexP.Wait();         // wait for mutual exclusion
      bBuffer[in] = item;
      System.out.println("-> " + item.toString());          // tracing
      in = (in + 1) % N;
      mutexP.Signal();       // signal for mutual exclusion
      notEmpty.Signal();  // notify a waiting consumer, if any
   }
```

```
    public Object get() {
        notEmpty.Wait();    // wait until the buffer becomes not empty
        mutexG.Wait();      // wait for mutual exclusion
        Object item = bBuffer[out];
        System.out.println("   " + item.toString() + " ->");   // tracing
        out = (out + 1) % N;
        mutexG.Signal();    // signal for mutual exclusion
        notFull.Signal();   // notify a waiting producer, if any
        return item;
    }
}
```

Notice that we did not write the keyword **synchronized** in the above methods put(*item*) and get(), because mutual exclusion is indeed guaranteed by the semaphores *mutex_P* and *mutex_G*.

The BoundedBuffer class can be seen in action by using variants of the classes: (i) Producer, (ii) Consumer, and (iii) BoundedBufferMonitorTester that we have presented above when describing the Bounded Buffer monitor in Java in Section 5.3. These variant classes are obtained by replacing every occurrence of the string (or substring) 'BoundedBufferMonitor' in the classes presented above by the string (or substring) 'BoundedBuffer' (see the statements marked by // — (M)). We assume that these variant classes are all stored in a single file named BoundedBufferTester.java.

We also assume that the four files:
(i) BoundedBuffer.java,
(ii) BoundedBufferTester.java,
(iii) CountingSemaphore.java, which defines the CountingSemaphore class, and
(iv) BinarySemaphore.java, which defines the BinarySemaphore class,
are all stored in a single folder. Then, if we execute the commands:

```
javac BoundedBufferTester.java
java  BoundedBufferTester 3
```

we construct a bounded buffer *bBuffer* with size $N = 3$ and we get the following output:

```
-> 1
   1 ->
-> 2
-> 3
-> 4
   2 ->
-> 5
   3 ->
-> 6
   4 ->
-> 7
   ...
```

This output is equal to the one we have obtained in Section 5.3 in the case of the BoundedBufferMonitorTester class. This output shows that, after a few steps, the buffer

of size 3 gets full, and for each item which is taken from the buffer, a new item is put in it (for instance, after the item 4 is taken out from the buffer, the item 7 is put in it). This behaviour is due to the fact that the producer puts items in the bounded buffer at a faster rate (once every 10 ms) than the rate (once every 100 ms) at which the consumer takes items away from the bounded buffer.

5.7 Five Philosophers Problem in Java

In this section we present a Java monitor for solving the Five Philosophers Problem. This monitor guarantees:

(i) mutual exclusion,

(ii) the absence of deadlock, and

(iii) the absence of starvation.

We need the following interface Monitor, stored in the file Monitor.java:

```
public interface Monitor {
  public void takeforks(int i);
  public void releaseforks(int i);
}
```

and the following three classes:

(i) ForkMonitor, stored in the file ForkMonitor.java,

(ii) PhilosopherThread, stored in the file ForkMonitorTester.java, and

(iii) ForkMonitorTester, also stored in the file ForkMonitorTester.java:

```
public class ForkMonitor implements Monitor {
  private static int N;
  private static boolean [] freefork;
  public ForkMonitor(int N) {
    this.N = N;
    this.freefork = new boolean[N];
    for (int i=0; i<n; i++) this.freefork[i] = true;
  }
  public synchronized void takeforks(int i) {
    while (freefork[i] == false)
      try { wait();                  // availablefork[i].wait;
      } catch (InterruptedException e){};
    freefork[i] = false;
    while (freefork[(i + 1) % N] == false)
      try { wait();                  // availablefork[(i + 1) % n].wait;
      } catch (InterruptedException e){};
    freefork[(i + 1) % N] = false;
  }
```

```
    public synchronized void releaseforks(int i) {
      freefork[i] = true;
      notify();                          // availablefork[i].signal;
      freefork[(i + 1) % N] = true;
      notify();                          // availablefork[(i + 1) % n].signal;
    }
  }

  class PhilosopherThread extends Thread {
    private int id = 0;
    private static ForkMonitor forkMonitor = null;
    public PhilosopherThread(int id, ForkMonitor forkMonitor) {
      this.id = id;
      this.forkMonitor = forkMonitor;
    }
    public void run() {
      while (true) {
        try {
        sleep(15 + 20 * id);    // thinking          ——— (†)
        forkMonitor.takeforks(id);
        System.out.print(id + "  ");  // id: the eating philosopher
        sleep(6);                  // eating
        forkMonitor.releaseforks(id);
        } catch (InterruptedException e) {}
      }
    }
  }

  public class ForkMonitorTester {
    public static void main(String[ ] args) {
      final int N =Integer.parseInt(args [0]);
      ForkMonitor forkMonitor = new ForkMonitor(N);
      for (int id=0; id<5; id++) new PhilosopherThread(id, forkMonitor).start();
    }
  }
```

Notice that from a theoretical viewpoint, we could allow two threads to concurrently run the releaseforks(*id*) method. However, we cannot erase the keyword **synchronized** in the releaseforks(*id*) method, because if we call *notify*() or *wait*() within a method that is *not* synchronized, Java does compile the program but at run time it raises an exception named IllegalMonitorStateException.

Then if we execute the commands:

```
javac ForkMonitorTester.java
java  ForkMonitorTester 5
```

we construct 5 PhilosopherThread objects, from PhilosopherThread 0 to PhilosopherThread 4, and we get the following sequence of eating PhilosopherThreads (below, for reasons of simplicity, we will refer to these PhilosopherThreads as 'philosophers'):

0 1 2 0 3 4 1 0 2 0 3 1 0 2 4 1 0 3 2 0 1 ...

Notice that the eating frequency is higher for philosophers who have smaller *id*. In particular, philosopher 0 eats 7 times, philosopher 1 eats 5 times, philosopher 2 eats 4 times, philosopher 3 eats 3 times, and philosopher 4 eats twice. This phenomenon is due to the fact that the thinking time is longer for philosophers with higher *id* (see line (†) in the PhilosopherThread class).

We leave to the reader to show that the above ForkMonitor class guarantees: (i) mutual exclusion, (ii) absence of deadlock, and (iii) absence of starvation.

5.8 Queue Monitor in Java

In this section we present the implementation in Java of a monitor for queues based on linked lists. This monitor is realized as a class called QueueMonitor stored in the file QueueMonitor.java.

```
public class QueueMonitor {
  private class Element {          // Element is a member class
    private Object datum;
    private Element next;
  }
  private Element first = null;
  private Element last = null;
  public QueueMonitor(){           // constructor
    this.first = null;
    this.last = null;
  }
  public synchronized void enqueue(Object datum) {
    Element newElement = new Element();
    newElement.datum = datum;
    newElement.next = null;
    if (first == null){ first = newElement; }
    else { last.next = newElement; };
    last = newElement;
    notify();
  }
  public synchronized Object dequeue() {
    while (first == null)
      try {wait();
      } catch (InterruptedException e){}
    Object datum = first.datum;
    first = first.next;
    return datum;
  }
```

```
    public synchronized void print() {
      Element first1 = first;
      System.out.print("queue: < ");
      while (first1 != null) {
        System.out.print(first1.datum + " ");
        first1 = first1.next;
      };
      System.out.println("<");
    }
  }
```

We can see this QueueMonitor class in action by using the following QueueUser and ConcurQueueMonitorTester classes both stored in the file ConcurQueueMonitorTester.java:

```
  class QueueUser extends Thread {
    static QueueMonitor queue = null;
    public QueueUser(QueueMonitor queue){
      this.queue = queue;
    }
    public void run() {
      queue.print();
      queue.enqueue(new Integer(7));   queue.print();
      try { sleep(10);
      } catch {InterruptedException e) { };
      queue.enqueue(new Integer(8));   queue.print();
      queue.dequeue();   queue.print();
    }
  }
  public class ConcurQueueMonitorTester {
    public static void main(String[] args) {
      QueueMonitor queue = new QueueMonitor();
      new QueueUser(queue).start();
      new QueueUser(queue).start();
    }
  }
```

If we execute the commands:

```
javac ConcurQueueMonitorTester.java
java  ConcurQueueMonitorTester
```

the following classes are generated:

(i) QueueMonitor.class, (ii) QueueMonitor$Element.class, (iii) QueueMonitor$1.class, (iv) QueueUser.class, and (v) ConcurQueueMonitorTester.class, and we get the following output:

queue: < <
queue: < 7 <
queue: < 7 <
queue: < 7 7 <
queue: < 7 7 8 <
queue: < 7 8 <
queue: < 7 8 8 <
queue: < 8 8 <

In this output the '<' signs are used to indicate that a number enters the queue from the right end and exits the queue from the left end.

6 Concurrent Programs Based on Handshaking Communications

We will present the pure CCS calculus and the value-passing CCS by Prof. Milner [16]. CCS stands for *Calculus for Communicating Systems*.

6.1 Pure CCS Calculus

In this section we present the pure CCS calculus by Milner [16]. Here are the syntactic categories.

- Names: A (A is a given set)
For every name ℓ we assume a co-name denoted by $\overline{\ell}$. The set of co-names is $\overline{A}$.
We assume that $\overline{\overline{\ell}} = \ell$ for any $\ell \in A$.

- Labels: $A \cup \overline{A}$
ℓ ranges over Labels

- Actions: $Act = A \cup \overline{A} \cup \{\tau\}$ (τ is a distinguished element)
α, β, ... range over Act.
Often the actions in $A \cup \overline{A}$ are said to be the *visible actions*, and the action τ is said to be the *invisible action*. A function f from Act to Act is said to be a renaming function iff $f(\overline{\ell}) = \overline{f(\ell)}$ and $f(\tau) = \tau$.

- Identifiers
P ranges over Id (Id is a given set)

- Processes
p ranges over $Proc$ $\quad p ::= 0 \mid \alpha.p \mid \sum_{i \in I} p_i \mid p_1 \,|\, p_2 \mid p\backslash L \mid p[f] \mid P$

where: I is a set of indexes, $L \subseteq A$ is a set of names, and
f is a renaming function.

- Definitions
$$P =_{def} p$$

Processes will also be called *terms*. Process 0 can be viewed as syntactically identical to $\sum_{i \in I} p_i$, whenever $I = \emptyset$. $\sum_{i \in \{1,2\}} p_i$ is also written as $p_1 + p_2$. The operators $+$ and $|$ are commutative and associative.
In definitions we allow ourselves to write $=$, instead of $=_{def}$.

Now we define the operational semantics of processes by defining the following relation $\xrightarrow{\alpha} \subseteq Proc \times Proc$ for each $\alpha \in Act$.

Prefix (the process $\alpha.p$ can do the action α and becomes the process p):

$$\alpha.p \xrightarrow{\alpha} p$$

Sum:

$$\frac{p_j \xrightarrow{\alpha} q}{\sum_{i \in I} p_i \xrightarrow{\alpha} q} \qquad \text{if } j \in I$$

Composition (or parallel composition):

$$\frac{p_1 \xrightarrow{\alpha} p_1'}{p_1 \,|\, p_2 \xrightarrow{\alpha} p_1' \,|\, p_2} \qquad \frac{p_2 \xrightarrow{\alpha} p_2'}{p_1 \,|\, p_2 \xrightarrow{\alpha} p_1 \,|\, p_2'} \qquad \frac{p_1 \xrightarrow{\ell} p_1' \qquad p_2 \xrightarrow{\overline{\ell}} p_2'}{p_1 \,|\, p_2 \xrightarrow{\tau} p_1' \,|\, p_2'}$$

Restriction:

$$\frac{p \xrightarrow{\alpha} q}{p\backslash L \xrightarrow{\alpha} q\backslash L} \qquad \text{if } \alpha \notin L \cup \overline{L} \text{ for any set } L \subseteq A \text{ of names}$$

Relabelling:

$$\frac{p \xrightarrow{\alpha} q}{p[f] \xrightarrow{f(\alpha)} q[f]} \qquad \text{for any renaming function } f$$

Identifier:

$$\frac{p \xrightarrow{\alpha} q}{P \xrightarrow{\alpha} q} \qquad \text{where } P =_{def} p$$

Definition 1. If $A \xrightarrow{\alpha} A'$ we say that A' is an α-*derivative* of A for any $\alpha \in Act$.

We have that, for any action a and any process A and B,
(i) $a.A \,|\, \overline{a}.B \xrightarrow{a} A \,|\, \overline{a}.B$ (ii) $a.A \,|\, \overline{a}.B \xrightarrow{\overline{a}} a.A \,|\, B$, and (iii) $a.A \,|\, \overline{a}.B \xrightarrow{\tau} A \,|\, B$.
We also have that, for any action a and any process A and B,
$(a.A \,|\, \overline{a}.B)\backslash\{a\} \xrightarrow{\tau} A \,|\, B$, and neither a-derivative nor $\overline{a}$-derivative exists for the term $(a.A \,|\, \overline{a}.B)\backslash\{a\}$.

Let us define the following relations $\approx$ and $=$ on $Proc \times Proc$. The relation $\approx$ is called *bisimulation equivalence* or *bisimilarity*, and the relation $=$ is called *bisimulation congruence*, or *equality* when no confusion arises. They are defined as the largest relations satisfying the following properties:

$$P \approx Q \text{ iff } \forall \alpha \in Act.\ \text{(i)}\ \forall P'.\ \text{if } P \xrightarrow{\alpha} P' \text{ then } (\exists Q'.\, Q \xRightarrow{\widehat{\alpha}} Q' \text{ and } P' \approx Q') \text{ and}$$
$$\text{(ii)}\ \forall Q'.\ \text{if } Q \xrightarrow{\alpha} Q' \text{ then } (\exists P'.\, P \xRightarrow{\widehat{\alpha}} P' \text{ and } P' \approx Q')$$

$$P = Q \text{ iff } \forall \alpha \in Act.\ \text{(i)}\ \forall P'.\ \text{if } P \xrightarrow{\alpha} P' \text{ then } (\exists Q'.\, Q \xRightarrow{\alpha} Q' \text{ and } P' \approx Q') \text{ and}$$
$$\text{(ii)}\ \forall Q'.\ \text{if } Q \xrightarrow{\alpha} Q' \text{ then } (\exists P'.\, P \xRightarrow{\alpha} P' \text{ and } P' \approx Q')$$

---

where the relations $\xrightarrow{\alpha}$, $\xRightarrow{\alpha}$, and $\xRightarrow{\widehat{\alpha}}$ are defined as follows.
Let t be any sequence of elements in Act, that is, $t \in Act^*$ and ε be the empty sequence. By $\widehat{t}$ we denote the sequence obtained from t by erasing all τ's. We have that $\widehat{\tau^n} = \varepsilon$ for any $n \geq 0$. Since $\alpha \in Act$, we have that: if $\alpha = \tau$ then $\widehat{\alpha} = \varepsilon$ else $\widehat{\alpha} = \alpha$. By $(\xrightarrow{\tau})^*$ we denote the reflexive, transitive closure of $\xrightarrow{\tau}$, that is, for all processes P and Q, $P\,(\xrightarrow{\tau})^*\,Q$ iff there exists $n \geq 0$ such that $P\,(\xrightarrow{\tau})^n\,Q$. Then,

(i) when writing $P \xrightarrow{t} Q$ for $t = \alpha_1 \ldots \alpha_n$ and $n \geq 0$, we mean that
$P \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} Q$;

(ii) when writing $P \stackrel{t}{\Longrightarrow} Q$ for $t = \alpha_1 \ldots \alpha_n$ and $n \geq 0$, we mean that
$P\,(\xrightarrow{\tau})^*\,\xrightarrow{\alpha_1}\,(\xrightarrow{\tau})^* \ldots (\xrightarrow{\tau})^*\,\xrightarrow{\alpha_n}\,(\xrightarrow{\tau})^*\,Q$ and τ may occur in t; and

(iii) when writing $P \stackrel{\widehat{t}}{\Longrightarrow} Q$ for $t = \alpha_1 \ldots \alpha_n$ and $n \geq 0$, we mean that

$P\,(\xrightarrow{\tau})^*\,\xrightarrow{\beta_1}\,(\xrightarrow{\tau})^* \ldots (\xrightarrow{\tau})^*\,\xrightarrow{\beta_m}\,(\xrightarrow{\tau})^*\,Q$ where $\widehat{t} = \beta_1 \ldots \beta_m$, for $m \geq 0$, is obtained from t by erasing all τ's. Thus, for all processes P and Q, $P \stackrel{\varepsilon}{\Longrightarrow} Q$ iff $P\,(\xrightarrow{\tau})^n\,Q$ for some $n \geq 0$. In particular, for every process P, $P \stackrel{\varepsilon}{\Longrightarrow} P$.

Let a CCS *context* $C[\_]$ be a CCS term C *without a subterm*.
For instance $\alpha.\_$, $P + \_$, $\_ \mid P$, and $(\_ \mid P) + Q$ are CCS contexts. $(\_|\_)$ is not a CCS context.

We have that $\approx$ is an equivalence relation. $\approx$ is not a congruence, that is, there exist a context $C[\_]$ and two terms t_1 and t_2 such that $t_1 \approx t_2$ and $C[t_1] \not\approx C[t_2]$. Indeed, for some distinct actions a and b in Act, one can show that:

(i) $b.0 \approx \tau.b.0$ and (ii) $a.0 + b.0 \not\approx a.0 + \tau.b.0$.

The proofs are as follows.

(i) holds because $b.0 \xrightarrow{b} 0$ and $\tau.b.0 \stackrel{\widehat{b}}{\Longrightarrow} 0$ (because $\tau.b.0 \xrightarrow{\tau} \xrightarrow{b} 0$), and

(ii) holds because $a.0 + \tau.b.0 \xrightarrow{\tau} b.0$ and $a.0 + b.0 \stackrel{\widehat{\tau}}{\Longrightarrow} a.0 + b.0$ (and obviously, $b.0 \not\approx a.0 + b.0$).

One can show that $=$ is a congruence and it is the largest congruence contained in $\approx$. We leave as an exercise to the reader to show that:

$P \mid \tau.Q \approx P \mid Q$ and $P \mid \tau.Q \neq P \mid Q$.

The equivalence $\approx$ satisfies the following laws which hold for any process P, Q, and R, any action α and β, any set $L \subseteq A$ of names, and any renaming function f.

Monoid laws:

1. $P + Q \approx Q + P$
2. $(P + Q) + R \approx P + (Q + R)$
3. $P + P \approx P$
4. $P + 0 \approx P$

τ laws:

5. $P + \tau.P \approx \tau.P$
6. $\alpha.\tau.P \approx \alpha.P$
7. $\alpha.(P + \tau.Q) \approx \alpha.(P + \tau.Q) + \alpha.Q$

Restriction laws:

8. $0\backslash L \approx 0$
9. $(P + Q)\backslash L \approx P\backslash L + Q\backslash L$
10. $(\alpha.P)\backslash\{\beta\} \approx$ *if* $\alpha \in \{\beta, \overline{\beta}\}$ *then* 0 *else* $\alpha.(P\backslash\{\beta\})$

Renaming laws:

11. $0[f] \approx 0$
12. $(P + Q)[f] \approx P[f] + Q[f]$
13. $(\alpha.P)[f] \approx f(\alpha).(P[f])$

For any process P we have that $0 \,|\, P \approx P \,|\, 0 \approx P$.

Theorem 1. [Expansion Theorem] Let P be $\sum_{i\in I} \alpha_i.P_i$ and Q be $\sum_{j\in J} \beta_j.Q_j$. Then $P \,|\, Q \;\approx\; \sum_{i\in I} \alpha_i.(P_i \,|\, Q) + \sum_{j\in J} \beta_j.(P \,|\, Q_j) + \sum_{\alpha_i=\overline{\beta_j}} \tau.(P_i \,|\, Q_j)$.

Thus, every $|$ can be replaced by $+$. For instance, for every action a, b, a_1, a_2, b_1, and b_2 in Act, we have that:

(i) $a.0 \,|\, b.0 \approx a.b.0 + b.a.0$

(ii) $a_1.a_2.0 \,|\, b_1.b_2.0 \approx a_1.a_2.b_1.b_2.0 + a_1.b_1.a_2.b_2.0 + a_1.b_1.b_2.a_2.0 + b_1.a_1.a_2.b_2.0 + b_1.a_1.b_2.a_2.0 + b_1.b_2.a_1.a_2.0$

(iii) $a.0 \,|\, \bar{a}.0 \,|\, b.0 \approx \tau.b.0 + a.(\bar{a}.b.0 + b.\bar{a}.0) + \bar{a}.(a.b.0 + b.a.0) + b.(\tau.0 + a.\bar{a}.0 + \bar{a}.a.0)$

Equivalences (i) and (ii) tell us that, if two processes cannot perform a τ action together, then their parallel composition is reduced to the *interleavings* of their actions.

An interleaving of two sequences $\sigma_1 = \langle a_1, a_2, \ldots, a_n\rangle$ and $\sigma_2 = \langle b_1, b_2, \ldots, b_m\rangle$ is a sequence $\sigma = \langle c_1, c_2, \ldots, c_{n+m}\rangle$ such that: (i) for $i = 1, \ldots, n+m$, c_i is either an element of σ_1 or σ_2, (ii) if we erase all the a_i's from σ we get σ_2, and (iii) if we erase all the b_i's from σ we get σ_1.

Exercise 2. Show that: (i) for every process P, $0 \,|\, P \approx P$, and
(ii) for any process P and Q, $P + \tau.(P + Q) \approx \tau.(P + Q)$.
Solution of (ii). $P+\tau.(P+Q) \approx P+(P+Q)+\tau.(P+Q) \approx P+Q+\tau.(P+Q) \approx \tau.(P+Q)$.

We say that A is *stable* iff A has no τ-derivatives.
If $P \approx Q$ and P and Q are stable, then $P = Q$.

Definition 2. A process A is said to be *finite* iff it is of the form:

$A ::= 0 \mid \alpha.A \mid \sum_{i\in I} A_i$

and I is a finite set.

Below we list a system of seven axioms for $=$. These axioms hold for any process P, Q, and R, and any action α. The Monoid laws and the τ laws are analogous to those of the bisimulation equivalence we have listed above.

Monoid laws:

1. $P + Q = Q + P$
2. $(P + Q) + R = P + (Q + R)$
3. $P + P = P$
4. $P + 0 = P$

τ laws:

5. $P + \tau.P = \tau.P$
6. $\alpha.\tau.P = \alpha.P$
7. $\alpha.(P + \tau.Q) = \alpha.(P + \tau.Q) + \alpha.Q$

This system of axioms is *complete* for finite processes, that is, these axioms are sufficient for showing all equalities between finite processes.

We also have the following equalities which hold for any process P and Q, any action α and β, any set $L \subseteq A$ of names, and any renaming function f.

Restriction laws:

8. $0\backslash L = 0$
9. $(P + Q)\backslash L = P\backslash L + Q\backslash L$
10. $(\alpha.P)\backslash\{\beta\} = \mathit{if}\ \alpha \in \{\beta, \overline{\beta}\}\ \mathit{then}\ 0\ \mathit{else}\ \alpha.(P\backslash\{\beta\})$

Renaming laws:

11. $0[f] = 0$
12. $(P + Q)[f] = P[f] + Q[f]$
13. $(\alpha.P)[f] = f(\alpha).(P[f])$

The following theorem is analogous to Theorem 1 on the facing page.

Theorem 2. [Expansion Theorem] Let P be $\sum_{i\in I} \alpha_i.P_i$ and Q be $\sum_{j\in J} \beta_j.Q_j$. Then $P \mid Q = \sum_{i\in I} \alpha_i.(P_i \mid Q) + \sum_{j\in J} \beta_j.(P \mid Q_j) + \sum_{\alpha_i=\overline{\beta_j}} \tau.(P_i \mid Q_j)$.

Theorem 3. For any process P, Q, and R, any action α, any set L of names, and any renaming function f, if $P = Q$ we have that:
(i) $\alpha.P = \alpha.Q$, (ii) $P + R = Q + R$, (iii) $P \mid R = Q \mid R$, (iv) $P\backslash L = Q\backslash L$, and
(v) $P[f] = Q[f]$.

The following theorem relates bisimulation equivalence and equality.

Theorem 4. $P \approx Q$ iff $(P = Q$ or $P = \tau.Q$ or $\tau.P = Q)$.

Exercise 3. Show that: (i) for every process P, $0 \mid P = P$, and
(ii) for any process P and Q, $P + \tau.(P + Q) = \tau.(P + Q)$.

6.2 Verifying Peterson's Algorithm for Mutual Exclusion

Let us consider Peterson's algorithm for two processes P_1 and P_2 running in parallel. We follow the approach described in [21]. The general case for more than 2 processes is left to the reader.

We begin by presenting the encoding of a boolean variable as a CCS term. A boolean variable B holding the value *true* can be encoded as the CCS identifier Bt defined as follows:

$Bt =_{def} \overline{brt}.Bt + bwt.Bt + bwf.Bf$

A boolean variable B holding the value *false* can be encoded as the CCS identifier Bf defined as follows:

$Bf =_{def} \overline{brf}.Bf + bwt.Bt + bwf.Bf$

The idea behind this encoding is that a boolean variable B is a register that may output the value it stores through the read actions *brt* (short for: B is read and returns *true*) and *brf* (short for: B is read and returns *false*) or it may accept the write actions *bwt* (short for: B is written and becomes *true*) and *bwf* (short for: B is written and becomes *false*). According to the actions that the register performs, *either* it stays in its original state (which is either Bt or Bf) *or* it modifies its state (that is, it goes either from Bt to Bf or from Bf to Bt).

Similar encoding can be done for every variable which may take a *finite* number of values.

For the encoding of Peterson's algorithm we need two boolean variables q_1 and q_2, initialized to *false*, and an integer variable s which may take the value 1 or 2 only and is initialized either to 1 or 2. Recall that the processes P_1 and P_2 are defined as follows:

process P_1:	process P_2:
while *true* **do**	**while** *true* **do**
non-critical section 1;	non-critical section 2;
$q_1 := true$; $s := 1$;	$q_2 := true$; $s := 2$;
await $(\neg q_2) \vee (s = 2)$;	**await** $(\neg q_1) \vee (s = 1)$;
critical section 1;	critical section 2;
$q_1 := false$; **od**	$q_2 := false$; **od**

and they run concurrently. The definition of process P_2 is obtained from that of process P_1 by interchanging 1 and 2.

For the variable q_1 we have:

$$\begin{array}{ll} Q1t =_{def} & \overline{q1rt}.Q1t + q1wt.Q1t + q1wf.Q1f \\ Q1f =_{def} & \overline{q1rf}.Q1f + q1wt.Q1t + q1wf.Q1f \end{array}$$

For the variable q_2 we have:

$$\begin{array}{ll} Q2t =_{def} & \overline{q2rt}.Q2t + q2wt.Q2t + q2wf.Q2f \\ Q2f =_{def} & \overline{q2rf}.Q2f + q2wt.Q2t + q2wf.Q2f \end{array}$$

For the variable s we have:

$$\begin{array}{ll} S1 =_{def} & \overline{r1}.S1 + w1.S1 + w2.S2 \\ S2 =_{def} & \overline{r2}.S2 + w1.S1 + w2.S2 \end{array}$$

For the process P_1 we have the following three equations, collectively denoted by $(\dagger\, P1)$:

$$\left.\begin{array}{ll} P1 =_{def} & \overline{q1wt}.\overline{w1}.P11 \\ P11 =_{def} & q2rt.P11 + r1.P11 + q2rf.P12 + r2.P12 \\ P12 =_{def} & in.out.\overline{q1wf}.P1 \end{array}\right] (\dagger\, P1)$$

For the process P_2 we have the following three equations, collectively denoted by $(\dagger\, P2)$:

$$\left.\begin{array}{ll} P2 =_{def} & \overline{q2wt}.\overline{w2}.P21 \\ P21 =_{def} & q1rt.P21 + r2.P21 + q1rf.P22 + r1.P22 \\ P22 =_{def} & in.out.\overline{q2wf}.P2 \end{array}\right] (\dagger\, P2)$$

The initial value of q_1 and q_2 is *false* and we take the initial value of s to be 1.

In the above CCS definitions of the process P_1 and P_2, the critical sections are encoded by the two sequence of actions '$in.out$', while the non-critical sections are not encoded. Indeed, in order to prove the properties of Peterson's algorithm we are interested in, there is no need to encode the non-critical sections and, as the reader may convince himself, these non-critical sections may be incorporated in the actions $\overline{q1wt}$ and $\overline{q2wt}$, respectively.

The fact that Peterson's algorithm satisfies the properties of mutual exclusion and absence of deadlock can be shown by proving the following equivalence:

$$R \approx (P1 \,|\, P2 \,|\, Q1f \,|\, Q2f \,|\, S1)\backslash L \qquad (\dagger)$$

where: (i) $L = \{q1rt, q1rf, q1wt, q1wf,\ q2rt, q2rf, q2wt, q2wf,\ r1, r2, w1, w2\}$ and
(ii) the process R is defined as follows:

$R =_{def} in.out.R$ (mutual exclusion and absence of deadlock)

The above equivalence (†) holds with $S2$, instead of $S1$, and also with $S1 + S2$, instead of $S1$, because the initial value of the variable s is insignificant for the validity of that equivalence.

The equation for $P11$, and analogously the equation for $P21$, requires some explanation. When $q_2 = \mathit{false} \vee s = 2$ holds, process P_1 goes from $P11$ to $P12$, as desired. However, process P_1 remains at $P11$ when $q_2 = \mathit{true} \vee s \neq 2$ holds, while it should do so when $\neg\,(q_2 = \mathit{false} \vee s = 2)$ (that is, $q_2 = \mathit{true} \wedge s \neq 2$) holds. Thus, our equation for $P11$ makes process P_1 to stay in $P11$ more often than expected. This forces the presence of an extra τ-loop in the operational semantics of $(P1 \,|\, P2 \,|\, Q1f \,|\, Q2f \,|\, S1)\backslash L$. Nonetheless, this extra τ-loop is irrelevant because for every action a, process A, and process P such that $P =_{def} \tau.P + a.A$, we have that: $P \approx a.A$.

Notice that, if we replace the equation $P11 =_{def} q2rt.P11 + r1.P11$ by the equation $P11 =_{def} q2rt.(r1.P11 + r2.P12) + r1.(q2rt.P11 + q2rf.P12)$, there is still the possibility that process P_1 stays in $P11$ more than expected. This happens, for instance, when q_2 from *true* becomes *false* (i.e., P_2 exits its critical section and enters its non-critical section) during the interval of time between the action $q2rt$ and the action $r1$ of the subterm $q2rt.(r1.P11 + r2.P12)$, i.e., the interval of time between the reading of the value of q_2 (which is *true*) and the reading of the value of s (which is 1).

Remark 1. From our discussion above, it follows that if in the initial definitions $(\dagger\, P1)$ and $(\dagger\, P2)$ of processes P_1 and P_2, we replace the equations:

$P11 =_{def} q2rt.P11 + r1.P11 \ +\ q2rf.P12 + r2.P12$
$P21 =_{def} q1rt.P21 + r2.P21 \ +\ q1rf.P22 + r1.P22$

by the equations:

$P11 =_{def} q2rt.(r1.P11 + r2.P12) \ +\ r1.(q2rt.P11 + q2rf.P12) \ +\ q2rf.P12 + r2.P12$
$P21 =_{def} q1rt.(r2.P21 + r1.P22) \ +\ r2.(q1rt.P21 + q1rf.P22) \ +\ q1rf.P22 + r1.P22$

we still have that: $R \approx (P1 \,|\, P2 \,|\, Q1f \,|\, Q2f \,|\, S1)\backslash L$. □

Remark 2. If in the initial definitions $(\dagger\, P1)$ and $(\dagger\, P2)$ of processes P_1 and P_2, we replace the equations:

$P12 =_{def} in.out.\overline{q1wf}.P1$
$P22 =_{def} in.out.\overline{q2wf}.P2$

by the equations:

$P12 =_{def} in1.out1.\overline{q1wf}.P1$
$P22 =_{def} in2.out2.\overline{q2wf}.P2$

and we replace $R =_{def} in.out.R$ by $R =_{def} in1.out1.R + in2.out2.R$, we have that:

$R \not\approx (P1 \,|\, P2 \,|\, Q1f \,|\, Q2f \,|\, S1)\backslash L$.

Now let us explain why $R \approx (P1 \,|\, P2 \,|\, Q1f \,|\, Q2f \,|\, S1)\backslash L$ does not hold. Let us first indicate how to construct a graph $G(p)$ for any given CCS process p:

(i) the nodes of $G(p)$ are the distinct equivalence classes in which the bisimulation equivalence $\approx$ partitions the set $N(p) = \{q \mid \exists t \in Act^*\ p \xrightarrow{t} q\}$ of processes, and
(ii) the arcs of $G(p)$ correspond to the operational semantics of CCS in the sense that in $G(p)$ there is an arc from node r to s with label α, denoted $(r) \xrightarrow{\alpha} (s)$, iff there exist a process $\bar{r}$ in r and a process $\bar{s}$ in s such that $\bar{r} \xrightarrow{\alpha} \bar{s}$ for some action $\alpha \in Act$.

Now let us consider the two graphs $G(R)$ and $G((P1 \mid P2 \mid Q1f \mid Q2f \mid S1)\backslash L)$ depicted in Figure 17 below. In that figure node 0 denotes the equivalence class of the process $(P1 \mid P2 \mid Q1f \mid Q2f \mid S1)\backslash L$. Since:
(i) $0 \xrightarrow{\tau} 1 \xrightarrow{\tau} 15 \xrightarrow{\tau} 12$,
(ii) in state 12 only the $in1$ action can be performed,
(iii) the only state Z such that $R \stackrel{\widehat{\tau\tau\tau}}{\Longrightarrow} Z$, that is, $R \stackrel{\varepsilon}{\Longrightarrow} Z$, is R itself, and
(iv) in state R both actions $in1$ and $in2$ can be performed,
we have that:
$R \not\approx (P1 \mid P2 \mid Q1f \mid Q2f \mid S1)\backslash L$. □

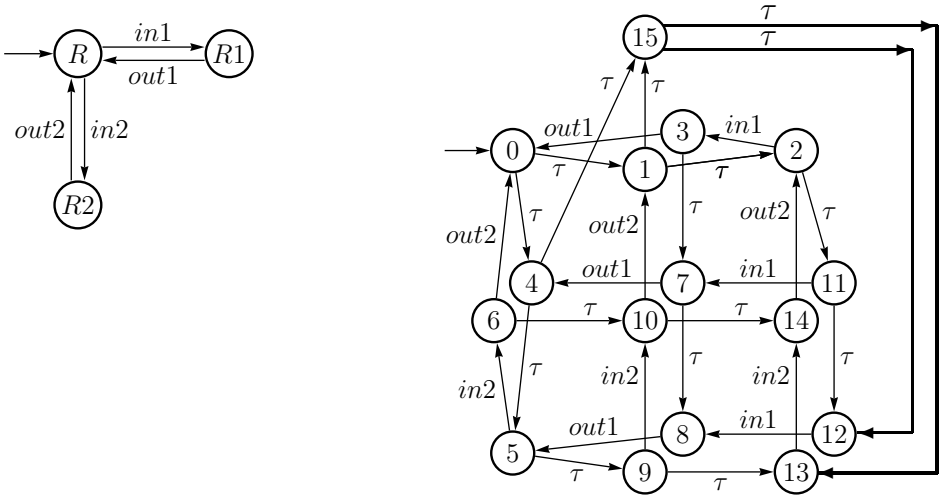


Fig. 17. Graphs associated with the processes R (left) and $(P1 \mid P2 \mid Q1f \mid Q2f \mid S1)\backslash L$ (right). Node 0 is the equivalence class of the process $(P1 \mid P2 \mid Q1f \mid Q2f \mid S1)\backslash L$.

When two processes compete for entering the critical section, Peterson's algorithm allows overtaking of one process w.r.t. the other process of *at most one* turn. More precisely, Peterson's algorithm ensures that:
(i) after executing '$q_2 := true$; $s := 2$;' process P_2 has to wait at most one execution of the critical section by process P_1, before P_2 enters its critical section and, symmetrically,
(ii) after executing '$q_1 := true$; $s := 1$;' process P_1 has to wait at most one execution of the critical section by process P_2, before P_1 enters its critical section.

This *bounded overtaking* property can be proved by showing the following bisimulation equivalence: $R_{bo} \approx (P1a \mid P2a \mid Q1t \mid Q2t \mid (S1{+}S2))\backslash L$ where:

$P1a =_{def}\ q2rt.P1a + r1.P1a\ +\ q2rf.P1b + r2.P1b$
$P1b =_{def}\ in1.out1.\overline{q1wf}.P1$

$P2a =_{def}\ q1rt.P2a + r2.P2a\ +\ q1rf.P2b + r1.P2b$
$P2b =_{def}\ in2.out2.\overline{q2wf}.P2$

$R_{bo} =_{def}\ \tau.in1.out1.in2.out2.R + \tau.in2.out2.in1.out1.R$ $\quad(\dagger R_{bo}1)$
$R\ \ =_{def}\ in.out.R$

and $P1$ and $P2$ are defined by the equations $(\dagger P1)$ and $(\dagger P2)$. Indeed, the definition of R_{bo} forces process $(P1a \mid P2a \mid Q1t \mid Q2t \mid (S1 + S2))\backslash L$ to perform sequences of visible actions which begin either by $in1.out1.in2.out2$ or by $in2.out2.in1.out1$, that is,
(i) after P_2 has executed '$q_2 := true$; $s := 2$;' and P_1 has left its critical section (that is, after $out1$ in the sequence $in1.out1.in2.out2$), P_2 enters its critical section before P_1 may enter again its critical section and, symmetrically,
(ii) after P_1 has executed '$q_1 := true$; $s := 1$;' and P_2 has left its critical section (that is, after $out2$ in the sequence $in2.out2.in1.out1$), P_1 enters its critical section before P_2 may enter again its critical section.

Note that: (i) the term $P1a$ denotes the state where process P_1 after executing '$q_1 := true$; $s := 1$;' is waiting to enter its critical section (that is, it is waiting to perform the action $in1$) and, symmetrically, (ii) the term $P2a$ denotes the state where process P_2 after executing '$q_2 := true$; $s := 2$;' is waiting to enter its critical section (that is, it is waiting to perform the action $in2$).

Note also that in the definition of R_{bo} above, we cannot erase any of the two τ's because, otherwise, the equivalence $R_{bo} \approx (P1a \mid P2a \mid Q1t \mid Q2t \mid (S1{+}S2))\backslash L$ would not hold. Indeed, the process $S1 + S2$ tells us that the initial value of the variable s can be either 1 or 2. We also have that $R_{bo2} \approx (P1a \mid P2a \mid Q1t \mid Q2t \mid S2)\backslash L$ where:

$R_{bo2} =_{def} in1.out1.in2.out2.R$ $\quad(\dagger R_{bo}2)$
$R\ \ =_{def} in.out.R$

Finally, we want to remark that in the above equation $(\dagger R_{bo}1)$ defining the process R_{bo}, we *cannot* replace R by R_{bo}, and still preserve the equivalence $R_{bo} \approx (P1a \mid P2a \mid Q1t \mid Q2t \mid (S1{+}S2))\backslash L$. Analoguously, in the above equation $(\dagger R_{bo}2)$ defining the process R_{bo2}, we *cannot* replace R by R_{bo2}, and still preserve the equivalence $R_{bo} \approx (P1a \mid P2a \mid Q1t \mid Q2t \mid S2)\backslash L$. These facts suggest that in order to show the bounded overtaking property, one may want to use a notion of equivalence which is weaker than $\approx$, such as the so called *trace equivalence*. For instance, we have that $a.b.0 + a.c.0$ and $a.(b.0 + c.0)$ are trace equivalent and yet it is *not* the case that $a.b.0 + a.c.0 \approx a.(b.0 + c.0)$. We leave the study of this issue to the interested reader.

Since in the equations $(\dagger P1)$ for process $P1$ we did *not* encode the non-critical section, process $P1$, immediately after exiting its critical section, makes a new request to enter again its critical section. The same fact holds for process $P2$. In order to disciplinate the accesses to the critical sections of the two processes, one may want the bounded overtaking property to hold for Peterson's algorithm and, indeed, that property holds. As a consequence, we have, for instance, that from state 15 of the graph of Figure 17 (in that state processes $P1$ and $P2$ are both executing the **await** statement because each of them may enter its critical section by performing a single τ action) it is impossible to have a sequence of actions that, forgetting the τ actions, is of the form $(in1.out1)^n$, for some $n > 1$. The only possible sequence of actions, forgetting the τ actions, begins by $in1.out1.in2.out2$ or by $in2.out2.in1.out1$.

In this section we have seen how properties of protocols can be shown by proving bisimulation equivalences. However, that it is not always simple to reduce the properties of interest to bisimulation equivalences. An alternative technique for proving properties of protocols can be found in [22]. It is a technique based on model checking. The interested reader may find the description of some more techniques in the relevant literature.

There are various automatic tools which can be used for checking whether or not two processes are bisimulation equivalent. We would like to suggest to the reader to use the Edinburgh-Sussex Concurrency Workbench [6]. This tool will give him the opportunity to perform some experiments and better understand the concepts we have introduced in this section.

6.3 Value-Passing CCS Calculus

In this section we present the value-passing CCS calculus by Milner [16]. Here are the syntactic categories.

- Variables
x ranges over *Vars* (*Vars* is a given set)
Variables get values over the integers $\mathbb{Z}$ (see Section 1).

- Arithmetic Expressions
a ranges over *AExpr* (see Section 1)

- Boolean Expressions
b ranges over *BExpr* (see Section 1)

- Channels
α ranges over *Chan* (*Chan* is a given set)
Let f be a renaming of Channels, that is, a bijection from Channels to Channels.

- Identifiers with arity
$P(x_1, \ldots, x_n)$ ranges over *Id* (*Id* is a given set)

- Processes
p ranges over *Proc*

$$p ::= 0 \mid \tau.p \mid \alpha!a \rightarrow p \mid \alpha?x \rightarrow p \mid b \rightarrow p \mid \sum_{i \in I} p_i \mid p_1 \,|\, p_2 \mid p\backslash L \mid p[f] \mid P(x_1, \ldots, x_n)$$

where: τ is a distinguished element, I is a set of indexes, and L is a set of channels.

- Definitions

$$P(x_1, \ldots, x_n) =_{def} p$$

Processes will also be called *terms*. The familiar construct *if* b *then* p_1 *else* p_2 is an abbreviation for $(b \rightarrow p_1) + (\neg b \rightarrow p_2)$.
$\alpha!a \rightarrow p$ means that the value of the arithmetic expression a is sent along the channel α and then the process $\alpha!a \rightarrow p$ becomes p. $\alpha?x \rightarrow p$ means that a value can be received along the channel α and if the value which is actually received is n then the process $\alpha?x \rightarrow p$ becomes $p[n/x]$, that is, the process p where all free occurrences of the identifier x have been replaced by n.

Instead of $\alpha?x \rightarrow p$ and $\alpha!a \rightarrow p$, we also write $\alpha x\,.\,p$ and $\overline{\alpha}a\,.\,p$, respectively. In definitions we allow ourselves to write $=$, instead of $=_{def}$.

Definitions of the form $P(x_1, \ldots, x_n) =_{def} p$ can be avoided by replacing the process $P(x_1, \ldots, x_n)$ by the process $rec\, P(x_1, \ldots, x_n).p$ where *rec* is a new constructor for processes [22].

Now we define the operational semantics of processes by defining the relation:
$\xrightarrow{\lambda} \subseteq Proc \times Proc$, for each $\lambda \in \{\alpha?x \mid \alpha \in Chan,\, x \in Vars\} \cup \{\alpha!n \mid \alpha \in Chan,\, n \in \mathbb{Z}\} \cup \{\tau\}$.
$a \to n$ means that the integer n is the value of the arithmetic expression a. $b \to true$ means that *true* is the value of the boolean expression b.

Guarded processes:

$$\tau.p \xrightarrow{\tau} p$$

$$(\alpha?x \to p) \xrightarrow{\alpha?n} p[n/x] \qquad \frac{a \to n}{(\alpha!a \to p) \xrightarrow{\alpha!n} p}$$

$$\frac{b \to true \qquad p \xrightarrow{\lambda} p'}{(b \to p) \xrightarrow{\lambda} p'}$$

Sum:

$$\frac{p_j \xrightarrow{\lambda} q}{\sum_{i \in I} p_i \xrightarrow{\lambda} q} \qquad \text{if } j \in I$$

Composition (or parallel composition):

$$\frac{p_1 \xrightarrow{\lambda} p_1'}{p_1 \,|\, p_2 \xrightarrow{\lambda} p_1' \,|\, p_2} \qquad \frac{p_2 \xrightarrow{\lambda} p_2'}{p_1 \,|\, p_2 \xrightarrow{\lambda} p_1 \,|\, p_2'}$$

$$\frac{p_1 \xrightarrow{\alpha?n} p_1' \qquad p_2 \xrightarrow{\alpha!n} p_2'}{p_1 \,|\, p_2 \xrightarrow{\tau} p_1' \,|\, p_2'} \qquad \frac{p_1 \xrightarrow{\alpha!n} p_1' \qquad p_2 \xrightarrow{\alpha?n} p_2'}{p_1 \,|\, p_2 \xrightarrow{\tau} p_1' \,|\, p_2'}$$

Restriction:

$$\frac{p \xrightarrow{\lambda} p'}{p\backslash L \xrightarrow{\lambda} p'\backslash L} \qquad \text{if for some value } n,\ \lambda \text{ is } \alpha?n \text{ or } \lambda \text{ is } \alpha!n \text{ then } \alpha \notin L$$

Relabelling:

$$\frac{p \xrightarrow{\lambda} p'}{p[f] \xrightarrow{f(\lambda)} p'[f]}$$

Identifier:

$$\frac{p \xrightarrow{\lambda} q}{P(x_1,\ldots,x_n) \xrightarrow{\lambda} q} \qquad \text{where } P(x_1, \ldots, x_n) =_{def} p$$

Recursion:

$$\frac{p[(rec\, P(x_1,\ldots,x_n).p)/P(x_1,\ldots,x_n)] \xrightarrow{\lambda} q}{rec\, P(x_1,\ldots,x_n).p \xrightarrow{\lambda} q}$$

The following laws and Theorem 5 are extensions to value-passing CCS of the corresponding laws and Theorem 1 which we have stated for pure CCS in Section 6.1. In these laws and theorem (i) p, q, and r denote processes, and (ii) $\gamma_i.p$ and $\delta_j.p$ denote any of the following: (ii.1) $\tau.p$, (ii.2) $\alpha?x \rightarrow p$, and (ii.3) $\alpha!a \rightarrow p$.

Monoid laws:

1. $p + q \approx q + p$
2. $p + (q + r) \approx (p + q) + r$
3. $p + p \approx p$
4. $p + 0 \approx p$

τ laws:

5. $\gamma.\tau.p \approx \gamma.p$
6. $p + \tau.p \approx \tau.p$
7. $\gamma.(p + \tau.q) + \gamma.q \approx \gamma.(p + \tau.q)$

The following theorem is analogous to Theorem 1 on page 88 and Theorem 2 on page 89.

Theorem 5. [Expansion Theorem] Let p be $\sum_{i\in I}\ \gamma_i.p_i$ and q be $\sum_{j\in J}\ \delta_j.q_j$. Then

$$\begin{array}{rl} p \,|\, q \ \approx & \sum_{i\in I}\ \gamma_i.(p_i \,|\, q)\ +\ \sum_{j\in J}\ \delta_j.(p \,|\, q_j)\ + \\ & \sum_{\gamma_i=\alpha?x,\ \delta_j=\alpha!a,\ a\rightarrow n}\ \tau.(p_i[n/x] \,|\, q_j)\ + \\ & \sum_{\gamma_i=\alpha!a,\ \delta_j=\alpha?x,\ a\rightarrow n}\ \tau.(p_i \,|\, q_j[n/x]). \end{array}$$

6.4 Verifying the Alternating Bit Protocol

Let us suppose that we have a finite stream $x_1, x_2, \ldots,\ x_n$ of data. A sender S wants to deliver this finite stream to a receiver R by making use of both the medium M_1 from S to R and the medium M_2 from R to S. We assume that each medium is unreliable in the sense that it may *loose* messages or *duplicate* them, but loss of messages and duplication of messages can occur a finite number of times only. More formally, we assume that for every *finite* sequence $\sigma = m_1, \ldots, m_k$ of messages which enters a medium, there exists a new, finite sequence σ' which exits from the medium and σ' is obtained from σ by replacing one m_i by a *finite* sequence of zero or more m_i's.

Notice that we do not allow a medium to *corrupt* messages, that is, a message m_i of the sequence σ cannot be replaced by a message different from m_i. (Actually, m_i can be replaced by the message m_{i-1} or m_{i+1}, because this replacement can be simulated by loss and duplication of messages.) Notice also that a medium is assumed to preserve the *order* of messages in the sense that if both m_i and m_j occur in σ' and $i < j$, then m_i occurs in σ' to the left of m_j.

The Alternating Bit protocol allows a reliable transmission of the stream $x_1, x_2, \ldots,\ x_n$ of data from S to R, even if messages may be lost or duplicated. In order to describe the Alternating Bit protocol we will find it useful to parameterize both the sender and the receiver, so that they will be denoted by $S(b)$ and $R(b)$, respectively, where the parameter b may be 0 or 1.

We need the following notation. If m is a pair of the form: $\langle x, b\rangle$ then $m1$ denotes x and $m2$ denotes b. Notice that if m is *error* then both $m1$ and $m2$ are undefined.

We define the unary operator $\neg$ by stipulating that: $\neg 1 = 0$ and $\neg 0 = 1$.

Then we define the following four CCS processes.

(i) A sender process (depicted in Figure 18):

$S(b) = inC\,x.\ S'(x,b)$

$S'(x,b) = \overline{inM_1}\,\langle x,b\rangle.\ outM_2\,y.\ \mathit{if}\ y = b\ \mathit{then}\ S(\neg b)\ \mathit{else}\ S'(x,b)$

The initial value of b is 1, thus initially, the sender process is $S(1)$.

(ii) A medium process M_1 from the sender $S(b)$ to the receiver $R(b)$:

$M_1 = inM_1\,m.\,(\overline{outM_1}\,m.\,M_1 + \overline{outM_1}\,error.\,M_1)$

(iii) A medium process M_2 from the receiver $R(b)$ to the sender $S(b)$:

$M_2 = inM_2\,y.\,(\overline{outM_2}\,y.\,M_2 + \overline{outM_2}\,error.\,M_2)$

(iv) A receiver process $R(b)$ (depicted in Figure 18):

$R(b) = \overline{inM_2}\,\neg b.\ outM_1\,m.\ \mathit{if}\ m2 = b\ \mathit{then}\ \overline{outC}\,m1.\ R(\neg b)\ \mathit{else}\ R(b).$

The initial value of b is 1, thus initially, the receiver process is $R(1)$.

In state $S'(x,b)$ the sender sends along the medium M_1 the datum x together with a bit b in the form of a pair $\langle x,b\rangle$. The bit b may be either 0 or 1. In the definition of the medium M_1 or M_2 the *error* message simulates the loss of a message. In the state $R(b)$ the receiver sends along the medium M_2 the bit $\neg b$.

The sender and the receiver processes can also be described by the following imperative procedures (see also Figure 18).

Sender $S(b)$:

σ_1: receive x from inC;
σ_2: send $\langle x,b\rangle$ to inM_1;
receive y from $outM_2$;
if $y = b$ *then begin* $b := \neg b$; goto σ_1 *end*;
goto σ_2

Receiver $R(b)$:

ρ: send $\neg b$ to inM_2;
receive m from $outM_1$;
if $m2 = b$ *then begin* send $m1$ to $outC$; $b := \neg b$ *end*;
goto ρ

We can show that the Alternating Bit protocol allows a reliable transmission of the stream $x_1, x_2, \ldots, x_n$ of data from S to R by showing that the system:

$SYST =_{def} (S(1) \mid M_1 \mid M_2 \mid R(1))\backslash\{inM_1, outM_1, inM_2, outM_2\}$

(depicted in Figure 19) is bisimulation equivalent to the process C defined as follows:

$C =_{def} inCx.\,\overline{outC}\,x.\,C$

Process C denotes the behaviour of a perfect channel which delivers in output every message which it accepts in input.

The equivalence $SYST \approx C$ can be established by using the laws and the theorem listed at the end of the previous section. The proof of the equivalence $SYST \approx C$ proceeds as follows. Let A denote the set of labels $\{inM_1, outM_1, inM_2, outM_2\}$.

$SYST \approx$

$\approx$ {*either* $S(1)$ communicates via inC *or* M_2 communicates with $R(1)$} $\approx$

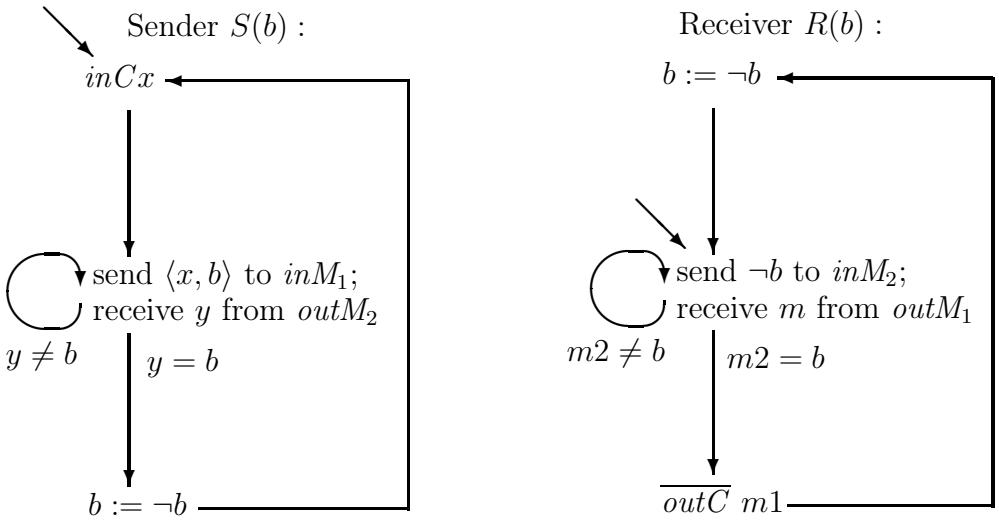


Fig. 18. Sender and receiver processes for the Alternating Bit protocol. Initially $b = 1$. The initial states are indicated with the incoming diagonal arrow.

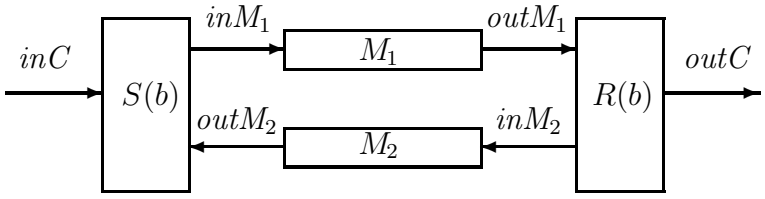


Fig. 19. The system $SYST$ for the Alternating Bit protocol. Initially, $b=1$.

$$
\begin{aligned}
&\approx inC\,x.(S'(x,1) \mid M_1 \mid M_2 \mid R(1))\backslash A + \\
&\quad \tau.(S(1) \mid M_1 \mid \overline{outM_2}\,0.M_2 + \overline{outM_2}\,error.M_2 \mid \\
&\qquad \mid outM_1\,m.\ if\ m2{=}1\ then\ \overline{outC}\,m1.\,R(0)\ else\ R(1))\backslash A \approx \\
&\approx \{either\ S'(x,1)\ \text{communicates with}\ M_1 \\
&\quad or\ M_2\ \text{communicates with}\ R(1) \\
&\quad or\ S(1)\ \text{communicates via}\ inC\} \approx \\
&\approx inC\,x.\tau.(\underline{outM_2}y.\ if\ y{=}b\ then\ S(\neg b)\ else\ S'(x,b) \mid \\
&\qquad \mid \overline{outM_1}\,\langle x,b\rangle.\,M_1 + \overline{outM_1}\,error.\,M_1 \mid M_2 \mid R(1))\backslash A + \\
&\quad inC\,x.\tau.(S'(x,1) \mid M_1 \mid \overline{outM_2}\,0.M_2 + \overline{outM_2}\,error.M_2 \mid \\
&\qquad \mid outM_1\,m.\ if\ m2{=}1\ then\ \overline{outC}\,m1.R(0)\ else\ R(1))\backslash A + \\
&\quad \tau.inC\,x.(S'(x,1) \mid M_1 \mid \overline{outM_2}\,0.M_2 + \overline{outM_2}\,error.M_2 \mid \\
&\qquad \mid outM_1\,m.\ if\ m2{=}1\ then\ \overline{outC}\,m1.\,R(0)\ else\ R(1))\backslash A \\
&\approx \ldots
\end{aligned}
$$

Instead of continuing this expansion, for reasons of simplicity, we will consider a smaller version of the system, called $SYST_{AB}$ (see Figure 20), where the media M_1 and M_2 are no longer present and, instead, the sender $S(b)$ and the receiver $R(b)$ are able to send correct

messages and *error*'s as well. The case in which an *error* is sent is also covering the case when the bit b of a message $\langle x, b\rangle$ is modified during transmission (that is, b becomes $\neg b$).

It will be much simpler to establish that $SYST_{AB} \approx C$, rather than $SYST \approx C$.

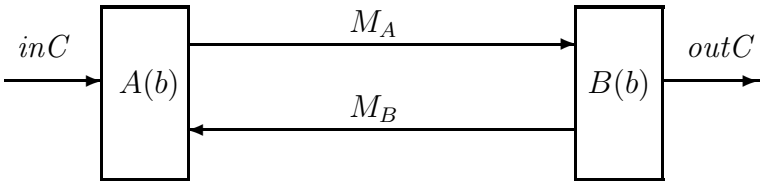


Fig. 20. The simplified system $SYST_{AB}$ for the Alternating Bit protocol with the sender $A(b)$ and the receiver $B(b)$. Initially b is 1.

Here are the new definitions of the sender and receiver processes, now called $A(b)$ and $B(b)$, respectively.

(i) Sender $A(b)$:

$A(b) = inC\, x.\ A'(x, b)$
$A'(x, b) = \overline{M_A}\, \langle x, b\rangle.\, A''(x, b) + \overline{M_A}\, error.\, A''(x, b)$
$A''(x, b) = M_B\, y.\ if\ y = b\ then\ A(\neg b)\ else\ A'(x, b)$

The initial value of b is 1 and, initially, the sender is $A(1)$.

(ii) Receiver $B(b)$:

$B(b) = M_A\, m.\ if\ m2 = b\ then\ \overline{outC}\, m1.\ B'(\neg b)\ else\ B'(b)$
$B'(b) = \overline{M_B}\, \neg b.\, B(b) + \overline{M_B}\, error.\, B(b).$

The initial value of b is 1 and, initially, the receiver is $B(1)$.

By expanding the system $SYST_{AB} = (A(1) \mid B(1))\backslash\{M_A, M_B\}$ we get a sequence of CCS terms which is depicted in the graph of Figure 21. In that figure we have adopted the following conventions:

(i) $\alpha \rightarrow m$ denotes the fact that the message (or bit) m is received from channel α,
(ii) $\alpha \rightarrow (m + n)$ denotes the fact that the message (or bit) m or n is received from channel α, and
(iii) $m \rightarrow \alpha$ denotes the fact that the message (or bit) m is sent out along channel α.

We leave it to the reader to show the equivalence $SYST_{AB} \approx C$. Notice that if the loops in Figure 21 are performed a finite number of times only, then a message which is sent by the sender $A(b)$ via channel M_A is received by the receiver $B(b)$ after a finite amount of time, in the sense that, for $b \in \{0, 1\}$, from the term $A'(x, b) \mid B(b)$ we derive the term $A''(x, b) \mid B'(\neg b)$ after a finite number of applications of the operational semantics rules. Analogously for messages sent by receiver $B(b)$ to the sender $A(b)$ via channel M_B.

The proof that $SYST_{AB} \approx C$ shows that the Alternating Bit protocol can cope with loss of messages, each loss being simulated by an *error* message output by the medium.

Remark 3. The reader may be unsatisfied by the fact that the equivalence $SYST_{AB} \approx C$ shows that the Alternating Bit protocol is able to cope with loss of messages, because

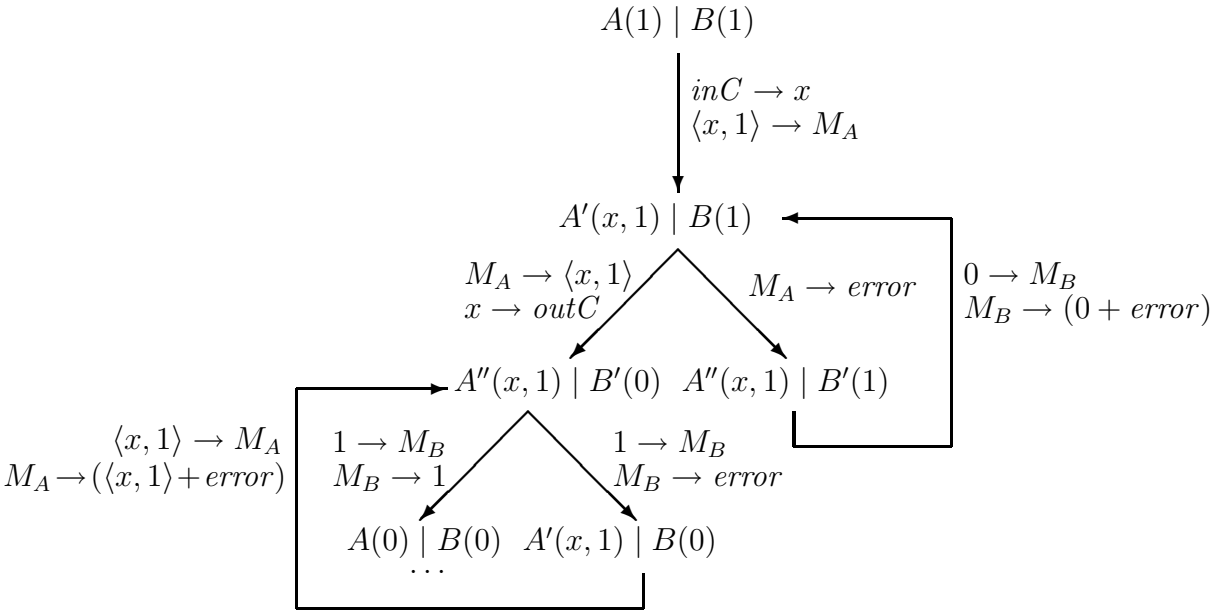


Fig. 21. The Alternating Bit protocol: Evolution of $SYST_{AB}$ when b is 1. In this figure all processes are assumed to be restricted w.r.t. $\{M_A, M_B\}$. The graph below node $A(0) \mid B(0)$ is like the graph below node $A(1) \mid B(1)$, except that 0 and 1 should be interchanged. When b is 0, the evolution of $SYST_{AB}$ is obtained from this figure by interchanging 0 and 1.

$SYST_{AB}$ contains τ-cycles (which corresponds to an indefinite loss and indefinite retransmission), while C does not contain τ-cycles. To satisfy the reader we may consider any one of the following three approaches.

(1) The first approach is to stipulate that the ability of the Alternating Bit protocol to cope with loss of messages is formalized by the equivalence $SYST_{AB} \approx C$ only in case the media M_1 and M_2 do not take the *error* option infinitely often, that is, there exists a finite loss of messages only (this is assumption is called *fairness assumption*).

(2) The second approach is to define the medium M_1 and M_2 as instances of the following medium M:

$M = inM\, m.\, \sum_{i \in \mathbb{N}} ErrM(i, m)$
$ErrM(0, m) = \overline{outM}\, m.\, M$
$ErrM(i+1, m) = \overline{outM}\, error.\, inM\, m'.\, ErrM(i, m')$

where $\mathbb{N}$ is the set of natural numbers. Thus, along that medium only finite loss of messages is possible.

(3) Finally, the third approach is to strengthen the definition of bisimilarity so that τ-cycles are not allowed. The interested reader may look at the discussion concerning this third approach in [16, pages 147, 166–169]. □

We leave to the reader to show that the Alternating Bit protocol can cope also with duplication of messages. Notice, however, that the Alternating Bit protocol cannot cope with corruption of messages.

7 Transactions and Serializability in Databases

7.1 Preliminaries

A database is a finite set D of data together with a finite set of operations on D.
Let *Loc* be a set of *locations* and *Val* a set of *values*. Each element $d \in D$ is called a *datum*. A datum d is a pair $\langle x, v\rangle$ of a location $x \in Loc$ and a value $v \in Val$. For each $x \in Loc$ there exists at most one value $v \in Val$, called the value of x, such that $\langle x, v\rangle \in D$. Thus, the set D of data is a finite, partial function from *Loc* to *Val*. We denote by $dom(D)$ the subset of *Loc* where the partial function D is defined, that is, $dom(D) = \{x \mid \langle x, v\rangle \in D\}$. Locations are also called *variables*.

A *transaction* is a sequence of (atomic or non-atomic) operations which act on the database. The atomic operations are the following ones:

input, *output*, *read*, *write*, assignment, *commit*, and *abort*.

The *input* and the assignment operations allow us to assign values to variables. The *output* operations allow us to print strings or print values of variables. The *read* and *write* operations behave as follows:

$read : Loc \to Val$	$read(x)$ returns the value $D(x)$ of the location x, if any. $read(x)$ is defined iff $x \in dom(D)$.
$write : Loc \times Val \to Void$	$write(x, v)$ associates the value v to the location x.

Void denotes the empty type and thus, a *write* operation does not return any value; it only produces side-effects (in our case, it changes the value of the store). A *commit* operation makes permanent the effects of the transaction on the database. An *abort* operation cancels the effects of the transaction on the database, that is, the database is left as it was before the transaction started.

Non-atomic operations are constructed from atomic operations as usual, by using the semicolon, **if-then-else,** and the **while-do** constructs. Thus, for instance, $a_1; a_2$ denotes the operation a_1 followed by the operation a_2. The **begin-end** construct is used for grouping operations.

The following examples show two transactions on a particular database which is a finite function from a set of account numbers (which are non-negative integers) to a set of amounts (which are non-negative integers).

Example 1. A deposit transaction. Given an account number *acc#* and an amount A, the following transaction deposits the amount A to the account number *acc#*. The transaction uses the local variable B.

procedure *deposit*(*acc#*, A);
  begin $B :=$ *read*(*acc#*);
    write(*acc#*, $B + A$);
    commit
  end

□

Example 2. A transfer transaction. Given in input two account numbers, say *from-acc#* and *to-acc#*, and an amount A, the following transaction transfers the amount A from the account number *from-acc#* to the account number *to-acc#*. The transaction uses the

local variable B and has no parameters. The values of the variables *from-acc#*, *to-acc#*, and A are provided via an *input* operation.

procedure *transfer*();
  begin *input*(*from-acc#*, A, *to-acc#*);
    $B :=$ *read*(*from-acc#*);
    if $B < A$ **then begin** *output*('insufficient funds'); *abort* **end**
    else begin *write*(*from-acc#*, $B - A$);
      $B :=$ *read*(*to-acc#*);
      write(*to-acc#*, $B + A$);
      output('transfer completed'); *commit*
      end
  end

□

Now we introduce two more operators, that is, | and +, for constructing non-atomic operations on databases. The operator | denotes *parallelism* and the operator + denotes *nondeterminism.* The semantics of parallelism is given by the nondeterministic choice among all possible *interleavings* of their atomic operations. We explain these notions by providing the following examples.
Case (1). For atomic operations a_1 and a_2, we state that $a_1 \,|\, a_2 = a_1; a_2 + a_2; a_1$.
Case (2). For non-atomic operations, say $(p_1; p_2)$ and $(p_3; p_4)$, we state that

$$(p_1; p_2) \,|\, (p_3; p_4) = (p_1; p_2; p_3; p_4) + (p_1; p_3; p_2; p_4) + (p_1; p_3; p_4; p_2) + (p_3; p_1; p_2; p_4) + (p_3; p_1; p_4; p_2) + (p_3; p_4; p_1; p_2).$$

In Case (1) there are two interleavings only: $a_1; a_2$ and $a_2; a_1$. In Case (2) there are six interleavings: $(p_1; p_2; p_3; p_4)$, $(p_1; p_3; p_2; p_4)$, $(p_1; p_3; p_4; p_2)$, $(p_3; p_1; p_2; p_4)$, $(p_3; p_1; p_4; p_2)$, and $(p_3; p_4; p_1; p_2)$.

When two or more transactions are performed in parallel and their atomic operations are sequentialized according to a particular interleaving, some problems may occur. The following example shows the occurrence of the so called *lost update* phenomenon.

Example 3. Two deposit transactions performed in parallel.
Let us assume that in the account number 15 there is the amount of 1000$. If two deposit transactions $T1$ and $T2$ acting on the same account number 15 are done in parallel, we may have the following undesirable *lost update* phenomenon. $B1$ is a local variable of the first deposit transaction and $B2$ is a local variable of the second deposit transaction.

time	$T1$: *deposit*(15,100$)	$T2$: *deposit*(15,200$)
0:	$B1 :=$ 1000$	
1:		$B2 :=$ 1000$
2:		*write*(15, $B2 +$ 200$)
3:		*commit*
4:	*write*(15, $B1 +$ 100$)	
5:	*commit*	

In the final state, at time 5, the database for the account number 15 has the amount of 1000$+100$, not the amount of 1000$+100$+200$, as we want to have. □

7.2 Serializability Theory

In order to be able to overcome problems like those due to lost updates we may use the results provided by the so called Serializabilty Theory. Let us begin the study of this theory by considering the following scenario indicating the way $n\,(\geq 2)$ transactions are performed *in parallel* on a database (see Figure 22 below). The *Transaction Manager* receives the atomic operations from these transactions, and then these atomic operations are ordered in a total order, one after the other, by the *Scheduler*. In that sense we say that the atomic operations of the transactions are *scheduled* by the Scheduler.

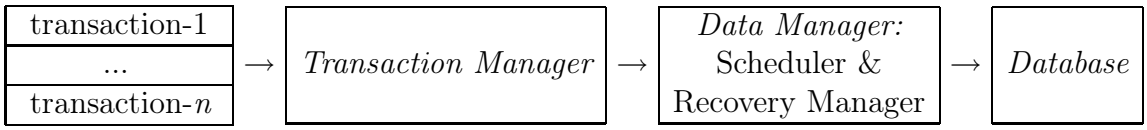


Fig. 22. Handling transactions on a database.

The *Recovery Manager* ensures that the effects on the database of the committed transactions (that is, those transactions which end with *commit*) are permanent, while the effects on the database of the aborted transactions (that is, those transactions which end with *abort*) are canceled. In order to restore the state of the database in case of an aborted transaction, we need to memorize, before executing any transaction t, the initial value of the variables which are used during that transaction t.

The Serializability Theory which we will study in the following sections, examines conditions and techniques which ensure that atomic operations of the various transactions which must be performed on a database are scheduled in a way that bad phenomena, such as the lost update phenomenon, do not occur.

7.3 An Abstract View of Transactions

In order to present the Serializability Theory we first consider the following *abstract view of transactions*.
(i) We replace *input* operations and assignment operations by *write* operations using new locations. For instance, the assignment $A := expr$ is replaced by $write(x_A, expr)$, where x_A is a new location associated with A. The variable x_A is then considered as a new element of the domain of the database.
(ii) *output* operations are discarded.
(iii) We also write c instead of *commit*, a instead of *abort*, $r(x)$ instead of $read(x)$, and $w(x)$ instead of $write(x, v)$. When we write $w(x)$, instead of $write(x, v)$, we abstract away from the value v. This abstraction does not influence the results of the serializability theory we present below.

Definition 1. An *abstract view of a transaction* (or a *transaction*, for short) is a pair $(T, <_T)$ where: (i) T is a finite set of atomic operations, and (ii) $<_T$ is an irreflexive, transitive binary relation on the set T such that the following conditions hold:
(i) $T \subseteq \{r(x) \mid x \in Loc\} \cup \{w(x) \mid x \in Loc\} \cup \{a, c\}$ (that is, only the atomic operations *read*, *write*, *commit*, and *abort* are considered),
(ii) $c \in T$ iff $a \notin T$ (and, thus, $c \notin T$ iff $a \in T$) (that is, a transaction is committed iff it is not aborted),
(iii) $\neg\exists t \in T$ $(a <_T t$ or $c <_T t)$ (that is, in a transaction no operation follows a *commit* or an *abort*), and
(iv) $\forall x \in Loc$ *if* $\{r(x), w(x)\} \subseteq T$ *then* either $r(x) <_T w(x)$ or $w(x) <_T r(x)$ (that is, a *read* and a *write* operation on the same location are related by $<_T$).

We will feel free to denote a given transaction $(T, <_T)$ by T only. The context will disambiguate between the two uses of the symbol T. We will also feel free to use the term 'transaction' both in the sense of Section 7.1 and in the sense of Definition 1 above.

The binary relation $<_T$ on the set T will also be called an *order* on the set T. For any two atomic operation t_1 and t_2 in T, we say that t_1 *precedes* t_2 iff $t_1 <_T t_2$. In Definition 1 we do not insist that $<_T$ be the minimal order satisfying Conditions (iii) and (iv) and, in particular, $<_T$ may relate more pairs than those which must be related by Condition (iv).

Example 4. The abstract views of the two transactions $T1$ and $T2$ of Example 3 of Section 7.1 are the pairs $(T1, <_{T1})$ and $(T2, <_{T2})$, respectively, defined as follows:

$T1 = \{w(x_{B1}), w(x_{15}), c\}$, $<_{T1} = \{\langle w(x_{B1}), w(x_{15})\rangle, \langle w(x_{15}), c\rangle, \langle w(x_{B1}), c\rangle\}$ and
$T2 = \{w(x_{B2}), w(x_{15}), c\}$, $<_{T2} = \{\langle w(x_{B2}), w(x_{15})\rangle, \langle w(x_{15}), c\rangle, \langle w(x_{B2}), c\rangle\}$

where the variables x_{B1} and x_{B2} stand for the local variables $B1$ and $B2$, respectively, and the variable x_{15} holds the balance of the account number 15. □

These abstract views $(T1, <_{T1})$ and $(T2, <_{T2})$ are depicted in Figure 23 below, where for any two given atomic operations op_i and op_j of a transaction T, $op_i <_T op_j$ is denoted by $op_i \rightarrow_T op_j$ or simply $op_i \rightarrow op_j$, when T is understood from the context. We omit to depict arrows which are entailed by transitivity of $<_T$. We adopt the same conventions also in the subsequent figures.

$$T1: \quad w(x_{B1}) \longrightarrow w(x_{15}) \longrightarrow c \qquad\qquad T2: \quad w(x_{B2}) \longrightarrow w(x_{15}) \longrightarrow c$$

Fig. 23. The abstract views of the transactions $T1$ and $T2$ of Example 3 of Section 7.1.

The relation $<_T$ describes the constraints under which the atomic operations of the transaction T should be scheduled. By this we mean that the atomic operations of T must be scheduled as a sequence $\sigma = \langle op_1, \ldots, op_n\rangle$ such that: (i) every atomic operation op_i in T occurs in σ, and (ii) for any two operations op_i and op_j in σ, if $op_i <_T op_j$ then $i < j$. This amounts to say that the scheduler must perform a topological sorting of T using the relation $<_T$.

Obviously, $<_T$ is *not* necessarily a linear order. Indeed, it may be the case that there exist two atomic operations op_i and op_j in T such that neither $op_i <_T op_j$ nor $op_j <_T op_i$ holds. This happens, for instance, in the case of the transaction:

$read(x)$; **if** p **then** $write(x, 1)$ **else** $write(y, 2)$; *commit*

whose abstract view, depicted in Figure 24, is $(T, <_T)$ where:

$T = \{r(x), w(x), w(y), c\}$ and
$<_T = \{\langle r(x), w(x)\rangle, \langle r(x), w(y)\rangle, \langle r(x), c\rangle, \langle w(x), c\rangle, \langle w(y), c\rangle\}$.

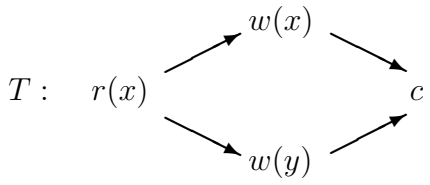


Fig. 24. A transaction T.

In the following Figure 25 we depicted two transactions which are used in the following Section 7.4.

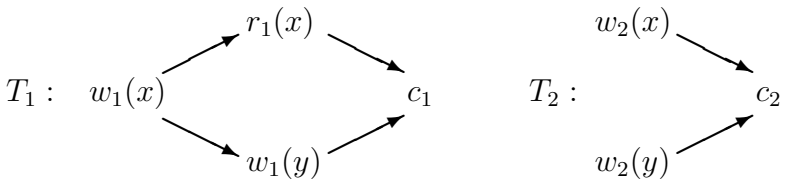


Fig. 25. The two transactions T_1 and T_2.

Remark 1. Notice that by Point (i) of Definition 1, we assume that in any given transaction and for any location x, there is at most one *read* operation $r(x)$ and one *write* operation $w(x)$ (recall that T is a set, *not* a multiset). If we want to allow more than one *read* operation or more than one *write* operation, we can do so by distinguishing these operations using subscripts. For instance, two *read* operations on the location x will be denoted by $r_1(x)$ and $r_2(x)$. Analogously, two *write* operations on the location x will be denoted by $w_1(x)$ and $w_2(x)$. As indicated in [3, page 26], it is possible to modify the Serializability Theory by allowing more than one *read* or *write* operation on the same location, so that the Serializability Theorem (see Theorem 6 below) continues to hold.

Remark 2. Now we show that is not a restriction to assume that in any given sequence of atomic operations generated by scheduling a transaction, for any location x, (i) there is at most one *read* operation $r(x)$, and (ii) there is at most one *write* operation $w(x)$.

We here present the proof of Property (i) and we leave to the reader the similar proof of Property (ii). Property (i) is shown by induction on the number of occurrences of the $r(x)$ operations.

Let us consider a given transaction T and a sequence σ of its atomic operations as it is generated by the scheduler when the transaction T is executed. Let us assume that σ has at least two $read(x)$ operations and it is of the following form:

$\sigma_1;\ B := read(x)\ ;\ \sigma_2;\ C := read(x)\ ;\ \sigma_3$

for some subsequences σ_1, σ_2, and σ_3 and no $read(x)$ operation occurs in σ_2. Thus, the abstract view $\overline{\sigma}$ of the sequence σ of the scheduled operations of the given transaction T is of the form:

$$\overline{\sigma_1}\,;\ r(x)\,;\ w(x_B)\,;\ \overline{\sigma_2}\,;r(x)\,;\ w(x_C)\,;\ \overline{\sigma_3} \qquad (\dagger)$$

where, for $i = 1, 2, 3$, $\overline{\sigma_i}$ is the abstract view of the sequence σ_i of operations, and the variables x_B and x_C hold the values of B and C, respectively. Unfortunately, in this sequence (†) we have at least two $r(x)$ operations.

(Case 1) Let us assume that in the subsequence σ_2 there is no *write* operation on the variable x. Then we can modify the scheduling of the given transaction T as follows: (i) we erase the assignment $C := read(x)$, and (ii) we replace C by B in σ_3. Since B and C have the same value, the resulting transaction performs the same changes on the database. Thus, as desired, the abstract view of the sequence of scheduled operations of T is of the form:

$\overline{\sigma_1}\,;\ r(x)\,;\ w(x_B)\,;\ \overline{\sigma_2}\,;\ \overline{\sigma_3}[w_B/w_C]$

where $\overline{\sigma_3}[w_B/w_C]$ is $\overline{\sigma_3}$ with w_B, instead of w_C. This resulting sequence has one occurrence of $r(x)$ less than the sequence (†).

(Case 2) Let us suppose that in the subsequence σ_2 there is at least one *write* operation on the variable x, and let $write(x, e)$, for some expression e, be the last one of these *write* operations. Then we can modify the given transaction T as follows: (i) we erase from σ_2 all *write* operations on x, except the last one, (ii) we replace this last operation by $write(y, e)$, where y is a new location, and (iii) we replace any subsequent reference to x by y (in particular, we replace $C := read(x)$ by $C := read(y)$). The modified transaction performs the same changes on the database as the original transaction T, modulo the renaming of x into y.

The abstract view of the sequence of the scheduled operations of the modified transaction is of the form:

$\overline{\sigma_1}\,;\ r(x)\,;\ w(x_B)\,;\ \overline{\sigma_{21}}\,;w(y)\,;\ \overline{\sigma_{22}}\,;r(y)\,;\ w(x_C)\,;\ \overline{\sigma_3}[y/x]$

where $\overline{\sigma_{21}}$ and $\overline{\sigma_{22}}$ are some subsequences of atomic operations, and $\overline{\sigma_3}[y/x]$ is $\overline{\sigma_3}$ with y, instead of x. This resulting sequence has one occurrence of $r(x)$ less than the sequence (†). □

7.4 Histories and Equivalent Histories

Let us consider a finite set $\{T_1, \ldots, T_n\}$ of transactions. For $i = 1, \ldots, n$, every atomic operation of T_i is given the subscript i. We also assume that all transactions $T_1, \ldots, T_n$ are committed, that is, for $i = 1, \ldots, n$, $c_i \in T_i$.

Definition 2. Let us consider a finite set $\{T_1, \ldots, T_n\}$ of transactions. We say that an operation p is *in conflict with* an operation q (or p and q are *conflicting*) iff $\exists i, j \in \{1, \ldots, n\}$ (with i and j not necessarily distinct) $\exists x \in Loc$ such that

either p is $r_i(x)$ and q is $w_j(x)$
or p is $w_i(x)$ and q is $w_j(x)$.

Definition 3. A *history* of the finite set $\{T_1, \ldots, T_n\}$ of transactions is the pair $(H, <_H)$, where the set H and the irreflexive, transitive order $<_H$ satisfy the following conditions:
(i) $H = \bigcup_i T_i$,
(ii) $<_H \supseteq \bigcup_i <_{T_i}$,
(iii) for any two conflicting operations $p, q \in H$, we have that either $p <_H q$ or $q <_H p$ (but not both, because $<_H$ is irreflexive).

We will feel free to denote a history $(H, <_H)$ by H only. The context will disambiguate between the two uses of the symbol H. Notice that in Definition 3 we do not insist that $<_H$ is the minimal order satisfying Conditions (ii) and (iii), that is, $<_H$ may relate more pairs of operations than those which must be related by Condition (iii).

Definition 4. Two histories $(H_1, <_{H_1})$ and $(H_2, <_{H_2})$ of the same set $\{T_1, \ldots, T_n\}$ of transactions are said to be *equivalent*, written $H_1 \approx H_2$, iff for each pair of conflicting operations, say p and q, they are *resolved in the same way*, that is, $\forall i, j \in \{1, \ldots, n\}$ $\forall p \in T_i$, $q \in T_j$ such that p is in conflict with q, we have that $p <_{H_1} q$ iff $p <_{H_2} q$.

Obviously, if p and q belong to the same transaction of the set $\{T_1, \ldots, T_n\}$, then it follows from Definition 3 that for all histories $(H_1, <_{H_1})$ and $(H_2, <_{H_2})$ of the set $\{T_1, \ldots, T_n\}$, $p <_{H_1} q$ iff $p <_{H_2} q$.

We leave it to the reader to show that $\approx$ is indeed an equivalence relation, that is, $\approx$ is reflexive, symmetric, and transitive.

In Figure 26 we show three histories, H_a, H_b, and H_c of the set $\{T_1, T_2\}$ of transactions depicted in Figure 25.

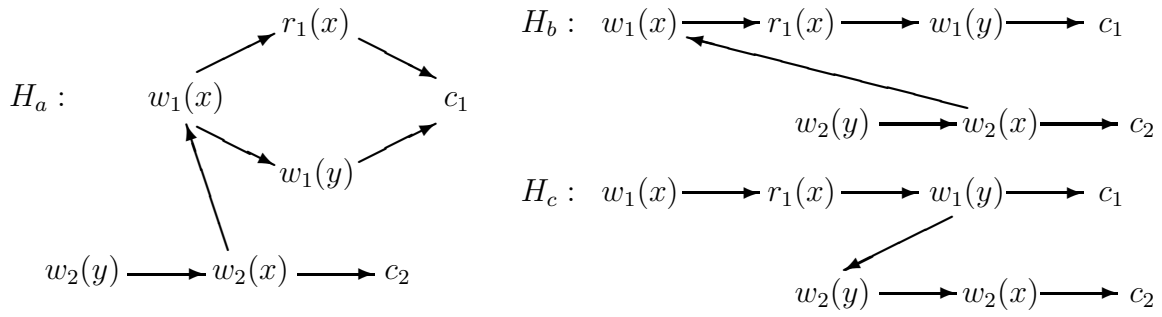


Fig. 26. The three histories H_a, H_b, and H_c of the set $\{T_1, T_2\}$ of transactions. The transactions T_1 and T_2 are depicted in Figure 25.

We have that $H_a \approx H_b$ and $H_a \not\approx H_c$ (thus, $H_b \not\approx H_c$).

7.5 The Serializability Theorem

Given two transactions T_1 and T_2, their *concatenation*, denoted $T_1 \cdot T_2$, is the irreflexive, transitive order which is the union of the following three sets:
- the irreflexive, transitive order relative to the transaction T_1,
- the irreflexive, transitive order relative to the transaction T_2, and
- $\{\langle a, b\rangle \mid a \in T_1,\ b \in T_2\}$.

Thus, the order relations within T_1 and T_2 are preserved and every operation of T_1 precedes every operation of T_2.

Definition 5. A history H of the finite set $\{T_1, \ldots, T_n\}$ of transactions is said to be *serializable* iff there exists a permutation $\langle i_1, \ldots, i_n\rangle$ of $\langle 1, \ldots, n\rangle$ such that H is equivalent to $T_{i_1} \cdot \ldots \cdot T_{i_n}$.

Notice that in the above Definition 5 we do not require that there exist *two* permutations $\langle i_1, \ldots, i_n\rangle$ and $\langle j_1, \ldots, j_n\rangle$ of $\langle 1, \ldots, n\rangle$ so that $T_{i_1} \cdot \ldots \cdot T_{i_n}$ is equivalent to $T_{j_1} \cdot \ldots \cdot T_{j_n}$.

Definition 6. Given a history H of the finite set $\{T_1, \ldots, T_n\}$ of transactions, the *serialization graph* of H, denoted $SG(H)$, is a directed graph such that: (i) the set of nodes is $\{T_1, \ldots, T_n\}$, and (ii) there is an edge from node T_i to node T_j iff $\exists$ an operation p of T_i and an operation q of T_j such that $p <_H q$ and p is in conflict with q.

Definition 7. The serialization graph $SG(H)$ of a history H of the finite set $\{T_1, \ldots, T_n\}$ of transactions is said to be *acyclic* iff there is no $i, j \in \{1, \ldots, n\}$, with $i \neq j$, such that in $SG(H)$ there is a directed path from T_i to T_j and a directed path from T_j to T_i.

Given the histories of Figure 26, the serialization graph of the histories H_a and H_b is: $T_2 \longrightarrow T_1$, while the serialization graph of the history H_c is: $T_1 \longrightarrow T_2$.

Theorem 6. (The Serializability Theorem) [3] A history H of the finite set $\{T_1, \ldots, T_n\}$ of transactions is serializable iff $SG(H)$ is acyclic.

As an example of application of the above Theorem 6 let us consider the three transactions of Figure 27 and the histories of Figures 28 and 29. We have that:
(i) $H_d \approx T_4 \cdot T_5$, (ii) $H_e \approx T_5 \cdot T_4$, and (iii) $H_f \approx H_g \approx H_h \approx T_6 \cdot T_4$,
while the histories H_i and H_j are not serializable, because:

$SG(H_i) = T_4 \rightleftarrows T_5$ and $SG(H_j) = T_4 \rightleftarrows T_6$.

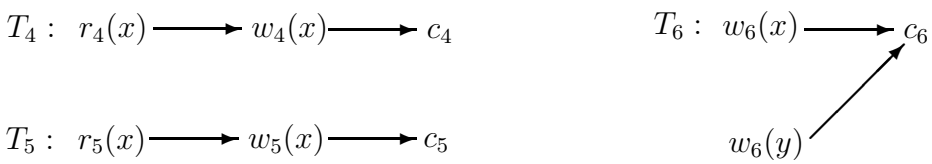


Fig. 27. The three transactions T_4, T_5, and T_6.

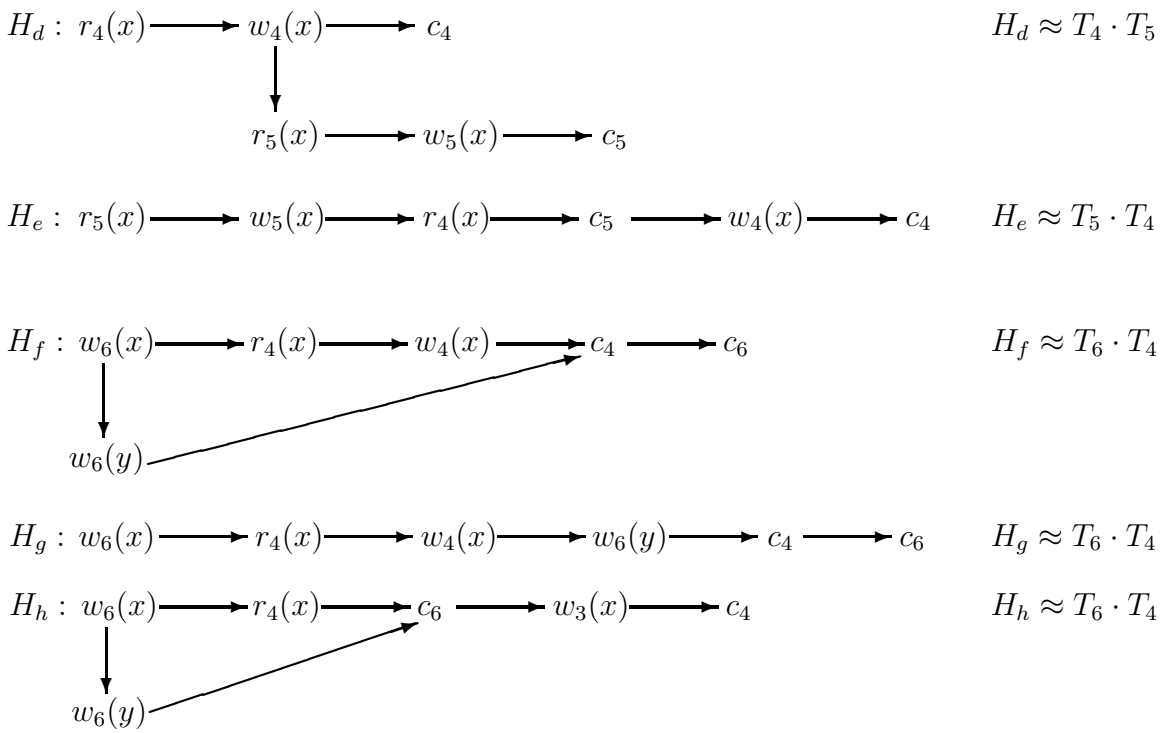


Fig. 28. Some serializable histories of the sets $\{T_4, T_5\}$ and $\{T_4, T_6\}$ of transactions. The transactions T_4, T_5, and T_6 are depicted in Figure 27.

The set of all histories of a finite set $\{T_1, \ldots, T_n\}$ of transactions is partitioned by the equivalence relation $\approx$ into a finite number of equivalence classes. Each equivalence class contains histories which are: (i) *either* serializable (and thus, they are associated to a particular permutation $\langle i_1, \ldots, i_n \rangle$ of $\langle 1, \ldots, n \rangle$, because they are all equivalent to the concatenation $T_{i_1} \cdot \ldots \cdot T_{i_n}$) (ii) *or* not serializable. Obviously, there are at most $n!$ equivalence classes of type (i) and there may be more than one equivalence class of type (ii).

Here is one more example of application of the above Theorem 6. Consider the two transactions T_7 and T_8 and the history H_k of Figure 30 on the next page. We have that history H_k is *not* serializable because we have that $SG(H_k) = T_7 \rightleftarrows T_8$. Indeed, the sequence $T_7 \cdot T_8$ of transactions may leave a wrong value in the location y and the sequence $T_8 \cdot T_7$ of transactions may leave a wrong value in the location x.

7.6 Locking Protocols

We assume that each location x has a *read lock* and a *write lock*. Before performing a *read* operation on a location x, the transaction T_i has to *obtain* the read lock of x. We also say that T_i has to *lock x for reading*. The operation of obtaining this lock is denoted by $rl_i(x)$ (rl stands for read lock, and the subscript i denotes the transaction). After performing the read operation, T_i *releases* the read lock which it has obtained, and we say

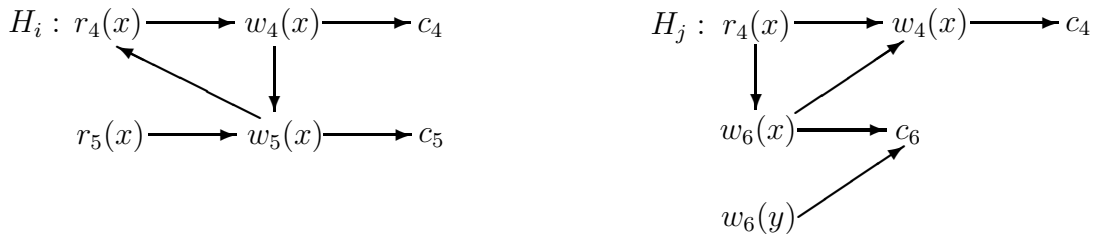


Fig. 29. The non-serializable histories H_i and H_j relative to the set of transactions $\{T_4, T_5\}$ and $\{T_4, T_6\}$, respectively. The transactions T_4, T_5, and T_6 are depicted in Figure 27.

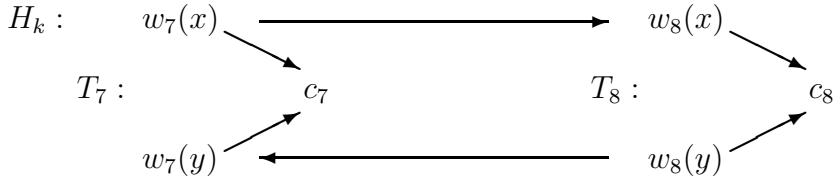


Fig. 30. The non-serializable history H_k relative to the set of transactions $\{T_7, T_8\}$.

that T_i *unlocks* x *after reading*. This operation of releasing the read lock of x is denoted by $ru_i(x)$ (ru stands for read unlock, and the subscript i denotes the transaction). Thus, instead of performing the read operation $r_i(x)$, the transaction T_i performs the sequence of operations: $rl_i(x)$; $r_i(x)$; $ru_i(x)$.

Analogously, before performing a *write* operation on x, the transaction T_i has to *obtain* the write lock of x. We also say that T_i has to *lock* x *for writing*. The operation of obtaining this lock is denoted by $wl_i(x)$ (wl stands for write lock). After performing the write operation, T_i *releases* the write lock which it has obtained, and we say that T_i *unlocks* x *after writing*. This operation of releasing the write lock of x is denoted by $wu_i(x)$ (wu stands for write unlock). Thus, instead of performing the write operation $w_i(x)$, the transaction T_i performs the sequence of operations:
$wl_i(x)$; $w_i(x)$; $wu_i(x)$.

All operations $rl_i(x)$, $r_i(x)$, $ru_i(x)$, $wl_i(x)$, $w_i(x)$, and $wu_i(x)$ are atomic.

Remark 3. The reader should not confuse the read locks and write locks we have now presented, with the locks used in Java for controlling the activities of threads. □

In the following Figure 31 we have depicted a diagram showing the acquisition and the release of locks while a transaction is performed. In that diagram the line goes up when a lock is obtained and goes down when a lock is released. The line is horizontal when a lock is neither obtained nor released and time elapses only because a read operation $r_i(x)$ or a write operation $w_i(x)$ is performed.

Notice that locks may be released in an order which is *not* consistent with the order of acquisition in the sense that, for instance, the sequence of the scheduled operations of a transaction, say T_i, may be $rl_i(x)$; $r_i(x)$; $wl_i(y)$; $w_i(y)$; $ru_i(x)$; $wu_i(y)$. That sequence may also be $rl_i(x)$; $r_i(x)$; $wl_i(y)$; $w_i(y)$; $wu_i(y)$; $ru_i(x)$.

Notice also that during a transaction a lock may be obtained after a lock has been released, and this allows the presence of a 'valley' in the diagram of Figure 31 (see the line between Point (1) and (2)).

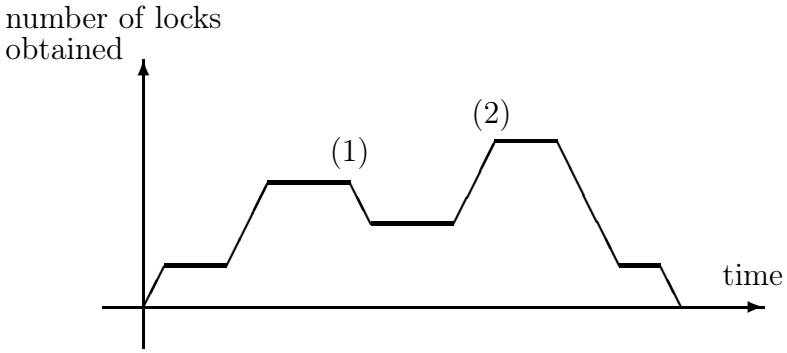


Fig. 31. Number of locks obtained (going up) and released (going down) during a transaction.

Now let us introduce the notion which defines when a locking operation is in conflict with another locking operation of a *different* transaction.

Definition 8. For any $x \in Loc$ and for any two different transactions T_i and T_j, the locking operation $wl_i(x)$ of T_i is *in conflict with* the locking operation $rl_j(x)$ of T_j and also with the locking operation $wl_j(x)$ of T_j.

Thus, a locking operation is in conflict with another locking operation iff (i) they are relative to the same location, (ii) they belong to two different transactions, and (iii) at least one of them is an operation to obtain a write lock. We stipulate that:

during the sequence $rl_i(x)$; $r_i(x)$; $ru_i(x)$ and the sequence $wl_i(x)$; $w_i(x)$; $wu_i(x)$ the scheduler may schedule other operations of different transactions, but these operations should not be in conflict with $rl_i(x)$ and $wl_i(x)$, respectively.

Remark 4. The reasons why the operation $r_i(x)$ has been replaced by the sequence of operations $rl_i(x)$; $r_i(x)$; $ru_i(x)$ and, analogously, the operation $w_i(x)$ has been replaced by $wl_i(x)$; $w_i(x)$; $wu_i(x)$, are the following ones:
(i) the mutual exclusion requirement for operations that are in conflict, and
(ii) the analogy with Peterson's algorithm for mutual exclusion.

Indeed, if the operation $r_i(x)$ is in conflict with another operation, say $w_i(x)$, these two operations $r_i(x)$ and $w_i(x)$ have to be scheduled one *after* the other, so that the variable x is accessed in a mutually exclusive way. If mutual exclusion is not guaranteed, the concurrent execution of $r_i(x)$ and $w_i(x)$ would produce an unpredictable result on the database. In order to realize mutual exclusion for operations that are in conflict, we act as in Peterson's algorithm which executes a so called *entry protocol* (or *trying protocol*) before the critical section and an *exit protocol* after the critical section. For instance, the entry protocol for process P_1 is (see Section 4.11):

$q_1 := true$; $s := 1$;
await $(\neg q_2) \vee (s = 2)$;

and the exit protocol is:

$q_1 := false$

For the operation $r_i(x)$ the entry protocol corresponds to the request $rl_i(x)$ of the read lock and the exit protocol corresponds to the release $ru_i(x)$ of the read lock. □

Locks can be realized by using *test-and-set* operations provided by the hardware. The execution of a *test-and-set*(x) operation is equivalent to 'we wait as long as x is *true*, and when we find (that is, *test*) that x is *false*, we make (that is, *set*) it *true*'. After setting x to *true*, the execution of *test-and-set*(x) terminates and we proceed.

We assume that *test-and-set*(x) operations are atomic, that is, no other operation can intervene between finding x to be *false* and setting its value to *true*. Unfortunately, in the realization of test-and set operations as we have described it, there is a form of *busy waiting*, that is, when a process executes a *test-and-set*(x) operation, it may perform again a test of x, even if x was not changed since it was last tested, and these repeated tests of x may occur an unbounded number of times if x never becomes *true*.

Locks can be realized by using a *test-and-set* operations, by taking advantage of the fact that after the execution of a *test-and-set* operation on a variable, say x, the value of x can be made available to all transactions and after reading that value, all transactions may determine their future operations so to realize restricted accesses to locations as the accesses allowed by the use of locks.

The reader may find more details on the realization of locks via *test-and-set* operations in Section 4.4 and in the relevant literature.

7.7 Two Phase Locking Protocols

A *Two Phase Locking protocol* (2PL protocol, for short) is a protocol for obtaining locks which obeys the following two rules.

(R1) When the scheduler receives a request $pl_i(x)$ to *obtain* a read lock or a write lock on location x from the transaction T_i (that is, $pl_i(x)$ is either $rl_i(x)$ or $wl_i(x)$), the scheduler tests whether or not $pl_i(x)$ is in conflict with any lock already given to *any other* transaction. If this is the case, the scheduler stops the transaction T_i, that is, it does not schedule any other atomic operation of T_i. Otherwise the scheduler gives to the transaction T_i the requested lock and then it may schedule subsequent atomic operations of the transaction T_i.

(R2) When a transaction has released a (read or write) lock, it cannot subsequently obtain any more (read or write) locks from the scheduler. □

A 2PL *scheduler* is a scheduler which adopts a 2PL protocol.

As a result of Rules (R1) and (R2), the sequence of the operations performed by each transaction according to a 2PL scheduler can be divided into two phases: (i) a *growing phase*, during which the transaction obtains locks, and (ii) a *shrinking phase*, during which the transaction releases locks.

These two phases give the name to the protocol.

In the following Figure 32 we have depicted a diagram showing the acquisition and the release of locks while a transaction is performed according to a Two Phase Locking protocol. Notice that no lock can be obtained after a lock has been released, that is, the line of the diagram does not go up after it started to go down for the first time. Thus, no valley is present in the diagram of Figure 32.

We have the following result.

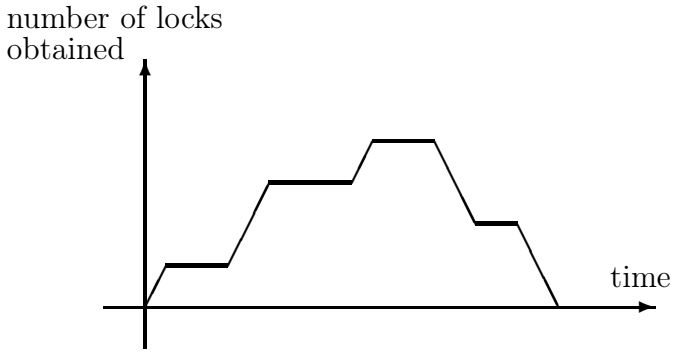


Fig. 32. Number of locks obtained (going up) and released (going down) during a transaction according to a Two Phase Locking scheduler.

Theorem 7. [3] Any history H which is constructed according to any 2PL protocol is serializable.

The result of this theorem can be promoted to an *iff* result. Indeed, a serializable history which is equivalent to the concatenation $T_1 \cdot \ldots \cdot T_n$ of transactions, can be constructed according to a 2PL protocol which, for any $i = 1, \ldots, n-1$, assigns the locks to the transaction T_{i+1} iff all operations of the transaction T_i have been performed.

Note that any history $(H, <_H)$ can be constructed by using a protocol, not necessarily a 2PL protocol, which gives the locks to the individual transactions of the history according to the topological sorting of their atomic operations following the order $<_H$. Obviously, $n\,(\geq 2)$ atomic operations can be done in parallel if: (i) they are not related by $<_H$, and (ii) n processors are available.

7.8 Strict Two Phase Locking Protocols

In practice, all implementations of 2PL protocols are variants of so called *strict* 2PL *protocols*.

A strict 2PL protocol is a 2PL protocol such that for every transaction T *all* locks which are obtained by T are released *together*, *after* T performs its *commit* or *abort* operation. More formally, this last condition can be expressed as follows:

if the atomic operations of a transaction T and the atomic operations of obtaining and releasing (read or write) locks are scheduled according to a sequence σ, and T obtains and releases (read or write) locks for the locations $x_1, \ldots, x_k$, *then* σ is of the form

α ; z ; $pu(x_1)$; ...; $pu(x_k)$

where: (i) α is a sequence of the atomic operations of T and the atomic operations for obtaining (read or write) locks, (ii) z is either c (*commit*) or a (*abort*), (iii) for all $j = 1, \ldots, k$, $pu(x_j)$ is either $ru(x_j)$ or $wu(x_j)$, that is, the atomic operation for releasing a (read or write) lock, and (iv) no atomic operations of other transactions are scheduled during the subsequence $pu(x_1)$; ...; $pu(x_k)$.

A scheduler which adopts a strict 2PL protocol is called a *strict* 2PL *scheduler*.

In the following Figure 33 we have depicted a diagram showing the acquisition and the release of locks while a transaction is performed according to a strict Two Phase Locking

protocol. Notice that after a lock has been released, no lock can be obtained and no read or write operation can be performed, that is, the line of the diagram cannot go up or stay horizontal after it started to go down for the first time. Thus, in the diagram of Figure 33 the line goes down from its highest level with a continuous descent without any horizontal stretch (see the line between Point (1) and (2)).

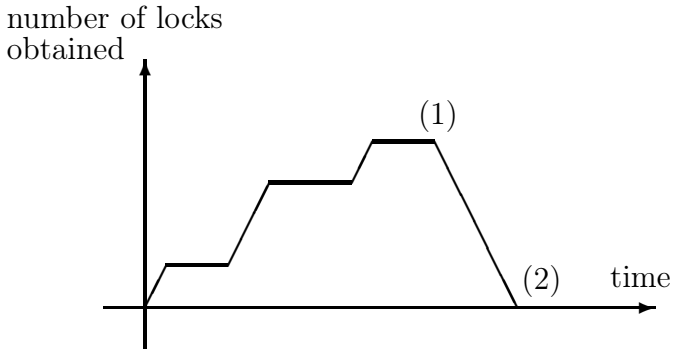


Fig. 33. Number of locks obtained (going up) and released (going down) during a transaction according to a *strict* Two Phase Locking scheduler.

7.9 Deadlocks

2PL schedulers and strict 2PL schedulers may determine *deadlocks*, that is, situations in which all transaction are stopped, as the following example shows.

Example 5. Let us consider the following two transactions:

$T_1 : r_1(x) \longrightarrow w_1(y) \longrightarrow c_1$, and
$T_2 : w_2(y) \longrightarrow w_2(x) \longrightarrow c_2$.

Let us consider the following sequence of operations: $rl_1(x)$; $r_1(x)$; $wl_2(y)$; $w_2(y)$ which can be generated by a 2PL scheduler or a strict 2PL scheduler. Then, the scheduler cannot schedule $wl_2(x)$ (for the operation $w_2(x)$) because it is in conflict with $rl_1(x)$. Thus, the transaction T_1 is stopped. Neither the scheduler can schedule $wl_1(y)$ (for the operation $w_1(y)$) because it is in conflict with $wl_2(y)$. Thus, also the transaction T_2 is stopped. Since both transactions are stopped, the remaining operations of T_1 and T_2 cannot be scheduled according to any 2PL scheduler (or any strict 2PL scheduler). □

In order to avoid deadlocks, the scheduler may *either* (i) abort a transaction which has not performed any operation for too long (this transaction is called the *victim transaction*), *or* (ii) keep a directed graph, called *waits-for graph*. The nodes of this graph are the transactions and for every i and j, with $i \neq j$, there is an edge from transaction T_i to transaction T_j iff T_i is waiting for T_j to release a lock, that is, T_i requests a lock which is in conflict with one which T_j has obtained and not released yet. There is a deadlock iff the waits-for graph has a cycle. To avoid a cycle is sufficient to abort one transaction in that cycle.

7.10 Conservative Two Phase Locking Protocols

It is possible to construct 2PL schedulers which never abort transactions. They are called *conservative* 2PL *schedulers* because they adopt particular 2PL protocols called *conservative* 2PL *protocols*. Conservative 2PL schedulers give to any transaction *all* its locks *before* any other operation of that transaction is performed. For these schedulers to work it is necessary that each transaction tells the scheduler its *readset* and its *writeset*: they are defined as follows.

The readset of a transaction T is the smallest set of locations such that a location x belongs to it if there exists an execution of T in which the operation $read(x)$ is performed. Analogously, the writeset of a transaction T is the smallest set of locations such that a location x belongs to it if there exist a value v and an execution of T in which the operation $write(x, v)$ is performed.

A conservative 2PL scheduler may delay operations *ahead of time*, that is, before than it is really necessary.

In the following Figure 34 we have depicted a diagram showing the acquisition and the release of locks while a transaction is performed according to a conservative Two Phase Locking protocol. Notice that first all locks are obtained, then read and write operations are performed, and finally, all locks are released, that is, the sequence of the operations of a transaction, say T_i, scheduled according to a conservative 2PL scheduler, is of the form:

$(rl_i(\ldots) + wl_i(\ldots))^*\,;\,(r_i(\ldots) + w_i(\ldots))^*\,;\,(ru_i(x)\ +\ wu_i(\ldots))^*$

Thus, in Figure 34 the line first goes up without any horizontal stretch, then having reached its highest level, it stays horizontal, and finally, it goes down without any other horizontal stretch.

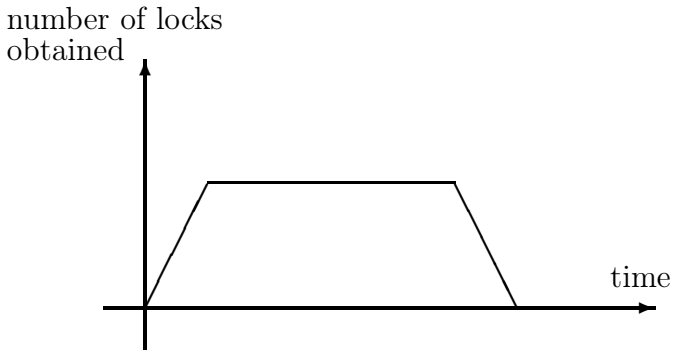


Fig. 34. Number of locks obtained (going up) and released (going down) during a transaction according to a *conservative* Two Phase Locking scheduler.

Remark 5. In some applications, given a set of transactions, we do not want the serializability of the histories of that set, and instead, we want the *mutual exclusion* of the various transactions, that is, the various transactions can be performed in any order we like as long as they access the data base in a mutually exclusive way. The reader may look at the previous Sections 4.5, 4.7, 4.8, and 4.11, where various techniques for ensuring mutual exclusion are presented. These techniques are based on constructs such as *semaphores*, *critical regions*, *conditional critical regions*, and *monitors.* □

8 Appendix: A Distributed Program for Computing Spanning Trees

In this appendix we present a sequential Java program which implements the distributed algorithm we have described in Section 4.12 on page 51 for computing a spanning tree of any given finite, undirected, connected graph represented as an array, called `node`, of `N` (>1) nodes. That array is defined as follows: `Node [] node = new Node [N]`. As indicated in Section 4.12, for `n`$\in\{$`0,...,N`–1$\}$, each node `n` of the graph is represented as a triple of arrays: `node[n].pArray`, `node[n].mArray`, and `node[n].sArray`, respectively. The triple representing the node `n` is denoted by $\langle P(n), M(n), S(n)\rangle$, when we refer to it outside the Java program. We assume that the nodes of the given graph are *left unmarked*.

Our sequential Java program is made out of the two classes `DistributedSpanningTree` and `ZeroOneGraph`, and implements the distributed algorithm consisting of the rewriting rules $\rho 1^{\Box}$, $\rho 2^{\Box}$, and $\rho 3^{\Box}$ listed below. That Java program implements the distributed algorithm in the sense that it satisfies the following property: for every execution of the Java program which makes a sequence σ of k (≥ 0) rule applications in the set $\rho 1^{\Box}; (\rho 2^{\Box} + \rho 3^{\Box})^*$, there exists an execution of the distributed algorithm which performs the same graph transformation by making a *partial order* π of k rule applications (indeed, some of these rule applications may be made in a concurrent way in different nodes of the graph) and π is consistent with the total order denoted by the sequence σ.

Our Java program implements the following rewriting rules which are applied in an *atomic way*: when a rule is applied to node n, no rule can be applied in any of the nodes in $P(n) \cup S(n)$. The root node is n_0. Note that rule $\rho 3^{\Box}$, instead of referring to the array E, may equivalently refer to the array M, because we assume that those two arrays are stored in the same locations.

$\rho 1^{\Box}$: For each node $p \in P(n_0)$, $S(p) \Rightarrow S(p) - \{n_0\}$ (the root has no father)
$\langle P(n_0), M(n_0), S(n_0)\rangle \Rightarrow \langle \{\}, \{\}, S(n_0)\rangle$
For each node $s \in S(n_0)$, $M(s) \Rightarrow \{n_0\}$ (sending $\otimes$ to each successor of the root)

$\rho 2^{\Box}$: Consider a node n such that there exists a node $i \in M(n)$ and (either $S(n) = \{\}$ or $P(n) \cap S(n) \neq \{\}$).
For each node $p \in P(n) - \{i\}$, $S(p) \Rightarrow S(p) - \{n\}$ (node i becomes father of node n)
$\langle P(n), M(n), S(n)\rangle \Rightarrow \langle \{i\}, \{\}, S(n)\rangle$ (erasing $\otimes$ arriving at node n)
For each node $s \in S(n)$, $M(s) \Rightarrow M(s) \cup \{n\}$ (sending $\otimes$ to each successor of node n)

$\rho 3^{\Box}$: Consider a node n such that $S(n) = E(n)$ (all sons of n have sent ▲ to n)
and there exists a node i such that $P(n) = \{i\}$. (node i is the father of node n)
$E(i) \Rightarrow E(i) \cup \{n\}$ (sending ▲ from node n to node i)

These rules are exactly those (with the same name) listed on page 56 except that the first components of the 4-tuples are dropped (and they became triples). Those first components can be dropped because: (i) they store the names of the nodes, and (ii) for those names we can use the indexes of the array of nodes that represents the graph.

The algorithm terminates iff all sequences of rule applications that can be generated by the algorithm terminate, and a sequence σ of rule applications terminates (or the

Termination Condition holds for σ) iff a finite (proper or not) prefix of σ leads to a situation where all sons of the root node n_0 have sent an end-token to the root node.

One can show that the Termination Condition holds for a sequence σ of rule applications if we assume that: (i) the Finite Delay hypothesis holds for σ, and (ii) the following two Conditions $(C1)$ and $(C2)$ hold.

Finite Delay Hypothesis for the sequence σ of rule applications:
if a (proper or not) prefix σ_1 of σ leads to a situation where a rule can fire at a node, then
(i) *either* for some *non-empty* subsequence σ_2 of rule applications we have that $\sigma = \sigma_1;\sigma_2$ (informally, if a rule can fire at a node, then there is a rule, maybe a different one, which actually fires at a node, maybe a different one, within a finite time),
(ii) *or* there is a (proper or not) prefix of σ for which the Termination Condition holds.

Condition $(C1)$: Rule $\rho 1^{\square}$ is applied to the root node n_0 only once and only at the beginning of the computation.

Condition $(C2)$: Rule $\rho 3^{\square}$ is applied to every unmarked node *at most once*.

Recall that this condition $C2$ can been enforced by using in rule $\rho 3^{\square}$ the property: $S(n) = E(n) \wedge \exists\, i,\ P(n) = \{i\} \wedge n \notin E(i)$, instead of the property: $S(n) = E(n) \wedge \exists\, i,\ P(n) = \{i\}$. Thus, in the Java program we have listed below, before applying $\rho 3^{\square}$ to the node `nRho3` we also make the test: `node[iRho3].mArray[nRho3] == 0` (this test checks that $n \notin E(i)$ holds).

In our program below we have ensured the Finite Delay hypothesis holds by realizing a sequence of rule applications which complies with the following metarule A:

apply $\rho 1^{\square}$ once to the root node; Metarule A
do { if there exists a node where $\rho 2^{\square}$ can be applied, then apply it once to that node;
if there exists a node where $\rho 3^{\square}$ can be applied, then apply it once to that node;
} while (not Termination Condition)

One can show that if there are more than two nodes in the given graph, the rules $\rho 2^{\square}$ and $\rho 3^{\square}$ are such that $\rho 3^{\square}$ cannot be applied to a node unless $\rho 2^{\square}$ has been already applied to that node (this is a consequence of the fact that we test that $P(n)$ is a singleton before applying $\rho 2^{\square}$ to node n).

The sequence of rule application complying with the above metarule A, simulates a sequence of rule applications which may occur in practice in a distributed algorithm for computing a spanning tree of a given graph.

Note that if we perform a sequence of rule applications which complies with the following metarule B, instead of metarule A:

apply $\rho 1^{\square}$ once to the root node; Metarule B
do { if there exists a node where $\rho 2^{\square}$ can be applied, then apply it once to that node;
do { if there exists a node where $\rho 3^{\square}$ can be applied, then apply it once to that node;
} while (there is a node where $\rho 3^{\square}$ can be applied)
} while (not Termination Condition)

then termination is *not* guaranteed, because for any integer k, there is a sequence of rule applications which: (i) complies with metarule B, (ii) it is longer than k, and (iii) at its end the Termination Condition does *not* hold.

This negative result follows from the fact that if rule $\rho 3^{\square}$ can be applied to some node, then it can be applied to that same node an unbounded number of times (these applications may be interleaved with applications of rule $\rho 3^{\square}$ to other nodes).

In our program we use the constructor `ZeroOneGraph(N, PerCentPROB)` for generating a random graph with `N` of nodes and probability `PerCentPROB` of arcs between nodes. That random graph is then transformed into one of its spanning trees by using the rules $\rho 1^{\square}$, $\rho 2^{\square}$, and $\rho 3^{\square}$.

```
/**
 * ========================================================================
 *
 *        DISTRIBUTED SPANNING TREE OF AN UNDIRECTED GRAPH
 *
 * N is the number of nodes of the finite, undirected, connected graph.
 * We assume that N > 1. The nodes are: 0, ..., N-1.
 * The initial node should be an element of {0, ..., N-1}.
 * ----------------------
 * Nodes are 3-tuples of arrays of N elements each:
 *    <pArray, mArray, sArray>.
 * The eArray has been stored in the same locations used by the mArray.
 * We know when that array is the mArray and when it is the eArray
 * by applying the conditions stated on Section 4.12 of [1].
 *
 * We identify the instructions where the mArray actually holds
 * the eArray by the note: // comment: mArray is eArray
 * Thus, we can do with 3-tuples of arrays, rather than 4-tuples.
 * ----------------------
 * Rule \rho1 is applied once only at the beginning of the computation.
 * Rule \rho2 can be applied once only at any node because it uses the
 * token in the mArray.
 * Rule \rho3 can be applied more than once at a node, but it is
 * idempotent, i.e., \rho3(\rho3 (node)) = \rho3 (node).
 * We could enforce that at every node \rho3 be applied once only
 * (see Section 4.12 of [1]).
 *
 * Termination occurs when end-tokens are sent to the initNode from
 * every son-node of the initNode.
 * ----------------------
 * Use the boolean variable 'traceon' to see the trace of the program.
 * ----------------------
```

```
 * [1] A. Pettorossi: Elements of Concurrent Programming. 3rd Edition.
 * Aracne 2009.
 * ========================================================================
 */
public class DistributedSpanningTree {

    static boolean traceon = false; // true: for tracing the execution
    static int N           = 0;     // to be read from the input: args[0]
    static int initNode    = 0;     // to be read from the input: args[1]
// ------------------------------------------------------------------------
// An object of the class Node is a node of the graph.
//
// The given graph is an array, named 'node' (unfortunately!),
// of N (>1) nodes, each of which is an object of the class Node.
// The name 'node' comes from the fact that, during the construction of
// the spanning tree, we manipulate the array 'node' so that at the end,
// it will denote a spanning tree of the given graph, and in a tree
// we usually identify a node with the subtree below that node.
// The identifier 'node' is declared as follows:
//       Node [] node = new Node [N];
// Every component of the array 'node' has three components, which are
// the arrays: (i) pArray, (ii) mArray, (iii) the sArray.
// -------------------
    static class Node {
      int [] pArray;                // 'predecessor nodes' of the node
      int [] mArray;                // 'marking nodes' of the node
                                    //  also 'end nodes' of the node
      int [] sArray;                // 'successor nodes' of the node

                                    // constructor
      public Node (ZeroOneGraph g) {
         this.pArray = new int [N];
         this.mArray = new int [N];
         this.sArray = new int [N];
      }
    }
// ------------------------ end of class Node -------------------------
// --------------------------------------------------------------------
// Method for printing the pArray, mArray, and sArray at every node.
// For instance, in order to print them after applying rule \rho3,
// use:        nodePrint(node,true,true,true)
// The three true's refer to the three arrays pArray, mArray, and sArray,
// respectively. Modify 'true' to 'false' to print the desired arrays.

    static void nodePrint(Node[] node, boolean p, boolean m, boolean s) {
       for (int i=0; i<N; i++) {
         if (p) {System.out.print("predecessors of node "+i+":  ");
                 for (int j=0; j<N; j++)
                     { System.out.print(node[i].pArray[j]+" ");};
                 System.out.print("\n");
                };
         if (m) {System.out.print("mArray of node "+i+":        ");
                 for (int j=0; j<N; j++)
                    { System.out.print(node[i].mArray[j] + " ");};
                 System.out.print("\n");
                };
```

```
         if (s) {System.out.print("successors of node "+i+":    ");
                 for (int j=0; j<N; j++)
                     { System.out.print(node[i].sArray[j]+" ");};
                 System.out.print("\n");
                };
         System.out.println();
       };
    }
// ---------------------------------------------------------------------
// Method for printing the spanning tree which is below the given
// node 'fromnode'
    static void treePrint(Node [] node, int fromNode) {
       for (int i=0; i<N; i++) {
           if ( node[fromNode].sArray[i] == 1 ) {
              System.out.println( fromNode + " -- " + i);
              treePrint(node, i);               }
       };
    }
// ---------------------------------------------------------------------
// Method for testing emptiness of an array

    static boolean emptyArray(int [] array) {
       boolean empty = true;
       for (int k=0; k<N; k++) {
           empty = empty && (array[k] == 0); };
       return empty;
    }
// ---------------------------------------------------------------------
// Method for computing the intersection of two arrays

    static int [] intersection(int [] aArray, int [] bArray) {
       int [] abArray = new int [N];
       for (int k=0; k<N; k++) {
           abArray[k] = aArray[k] * bArray [k]; };
       return abArray;
    }
// ---------------------------------------------------------------------
// Method for computing the equality of two arrays

    static boolean equal(int [] aArray, int [] bArray) {
       boolean eq = true;
       for (int k=0; k<N; k++) {
           eq = eq && (aArray[k] == bArray [k]); };
       return eq;
    }
// =====================================================================
//                            main method
   public static void main(String[] args) {
// ------------------------------ N is the number of Nodes of the graph.
// N should be greater than 1. The nodes are: 0, ..., N-1.

    try {N = Integer.parseInt(args[0]);
         if ( N < 2) { throw new RuntimeException (); };
    } catch (Exception e) {
                        System.out.println("*** Wrong number of nodes!");
                        System.exit(1);}
```

```
// ------------------------------ initNode: initial node.
// initNode is the root of the spanning tree. 0 <= initNode <= N-1.

    try {initNode = Integer.parseInt(args[1]);
         if ( ( 0 > initNode ) || ( initNode > N-1 ) ) {
             throw new RuntimeException ();
             };
    } catch (Exception e) {System.out.println("*** Wrong initial node!");
                           System.exit(1);}
// ========================================================================
// PerCentPROB: probability of connectivity. 0 < PerCentPROB <= 100.
// If PerCentPROB = k, every node is connected to the other N-1 nodes
// with probability k/100.
    final int PerCentPROB = 40;

// ------------
// Generation of a random graph with N nodes.
// Every node has (N-1) x PerCentPROB / 100 successor nodes (in average).
//
// graph[i][j] = 1 iff there is an arc from node i to node j.
// graph[i][j] = 0 iff there is no arc from node i to node j.

    ZeroOneGraph g = new ZeroOneGraph(N, PerCentPROB);
    g.graphPrint();
    System.out.println("Every node is connected to the other nodes " +
                       "with probability " + PerCentPROB + " %.");
    System.out.println("The initial node is: " + initNode + "\n");

//          'node' is the graph: it is an array of N objects
//            |                  each of which is of the class Node
//            V
    Node [] node = new Node [N];

// ------------
// INITIALIZATION OF THE GRAPH 'node'
// using the graph 'g' generated by the ZeroOneGraph constructor

     for (int i=0; i<N; i++) { node[i] = new Node(g);
                                         // predecessor nodes
        for (int j=0; j<N; j++) { node[i].pArray[j] = g.graph[j][i]; };
                                        // no mark-tokens: all 0's
        for (int j=0; j<N; j++) { node[i].mArray[j] = 0; };
                                        // successor nodes
        for (int j=0; j<N; j++) { node[i].sArray[j] = g.graph[i][j]; };
     };

// ========================================================================
// ====================== applying rule \rho1 to initNode

    // ------ for all p in P(n_0), S(p) => S(p)-{n_0}
    for (int p=0; p<N; p++) {
        if ( node[initNode].pArray[p] == 1 )
                { node[p].sArray[initNode] = 0; }
    };
    // ------ P(n_0) = {}
    for (int i=0; i<N; i++) { node[initNode].pArray[i] = 0; };
```

```
    // ------ for all s in S(n_0), M(s) => {n_0}
    for (int s=0; s<N; s++) {
      if (node[initNode].sArray[s] == 1) {node[s].mArray[initNode] = 1;};
    };
// ----------- printing after applying rule \rho1:
     if ( traceon ) { System.out.println(
                      " ---- Applying \\rho1 to node "+ initNode + "\n");
                      nodePrint(node,true,true,true); }
// ========================================================================
    boolean termination;         // boolean variable to check termination
// ------------------------------------------------------------------------
//
    do {                 // <--- 'do-while' for the rules \rho2 and \rho3

// ====================== applying rule \rho2 if there exists a node
//                         where it can be applied.
// It is the case that Rule \rho2 is applied at every node at most once.
// No extra condition to test is needed.

    boolean applyRho2 = false; // true iff rule \rho2 con be applied
                           // -1 denotes an undefined initial value
    int nRho2 = -1;        // node n where the rule \rho2 can be applied
    int iRho2 = -1;        // node i when the rule \fho2 is applied

    // -----------------
    searchApplyRho2:
    for (int n=0; n<N; n++) {               // <--- 'for' (of rule \rho2)
                               // S(n) = {}  or  P(n) \cap S(n) \not = {}
        if (emptyArray(node[n].sArray) ||
            !(emptyArray(intersection(node[n].sArray,node[n].pArray)))) {
                                            //  finding a node i in M(n)
            for (int i=0; i<N; i++) {
              if (node[n].mArray[i] == 1) {
               applyRho2 = true; nRho2 = n; iRho2 = i;
               break searchApplyRho2;
              };
            };
        };
    };                                 // <--- end of 'for' (of rule \rho2)
    // -----------------
    if ( applyRho2 ) {                          // <--- 'if' (of rule \rho2)

    // ------  for all p in P(n)-{i}, S(p) => S(p)-{n}
              for (int p=0; p<N; p++) {
                  if ( node[nRho2].pArray[p] == 1 && p != iRho2 )
                     { node[p].sArray[nRho2] = 0; }
              };

    // ------  P(n) = {i}
              for (int i=0; i<N; i++) { node[nRho2].pArray[i] = 0; };
              node[nRho2].pArray[iRho2] = 1;

    // ------  M(n) = {}
              for (int i=0; i<N; i++) { node[nRho2].mArray[i] = 0; };
```

```
    // ------  for all s in S(n), M(s) => M(s) U {n}
              for (int s=0; s<N; s++) {if ( node[nRho2].sArray[s] == 1 )
                  { node[s].mArray[nRho2] = 1; }
              };

// ----------- printing after applying rule \rho2:
         if ( traceon ) { System.out.println(
                          " ---- Applying \\rho2 to node "+ nRho2 +"\n");
                          nodePrint(node,true,true,true); }

    };                                  // <--- end of 'if' (of rule \rho2)

// ====================== applying rule \rho3 if there exists a node
//                        where it can be applied.
//
// Since rule \rho3 can fire infinitely many times at a node if it can
// fire once, in order to ensure termination, we apply rule \rho3
// exactly once to every node where it can be applied by testing the
// extra Condition (***): n \not \in E(i) (see below).

    boolean applyRho3;          // true iff rule \rho3 con be applied
                                // -1 denotes an undefined initial value
    int nRho3 = -1;             // node n where rule \rho3 can be applied
    int iRho3 = -1;             // node i for rule \rho3

    // -----------------
    searchApplyRho3:
    for (int n=0; n<N; n++) {               // <--- 'for' (of rule \rho3)
                                            // comment: mArray is eArray
                                            // S(n) = E(n)
    applyRho3 = equal(node[n].sArray, node[n].mArray);
    // ------ testing card == 1. One predecessor only for node nRho3:
    //                            \exists i. P(n) = {i}
    int card = 0;
    if (applyRho3) { nRho3 = n;
              for (int j=0; j<N; j++) {
                 if ( node[nRho3].pArray[j]==1 ) {card++; iRho3=j;}
              };
    };
    applyRho3 = (applyRho3 && card==1 && node[iRho3].mArray[nRho3]==0);
             // The condition: node[iRho3].mArray[nRho3] == 0   (***)
             // avoids more than one application of the rule \rho3 at the
             // same node. It is Condition (C2) in [1, Section 4.12]:
             //             n \not \in E(i)
    // -----------------
    if (applyRho3) {                         // <--- 'if' (of rule \rho3)
         node[iRho3].mArray[nRho3] = 1;      // E(i) => E(i) U {n}

// ----------- printing after applying rule \rho3:
         if ( traceon ) { System.out.println(
                          " ---- Applying \\rho3 to node "+ nRho3 +"\n");
                          nodePrint(node,true,true,true); };
     break searchApplyRho3;
     };                                 // <--- end of 'if'  (of rule \rho3)
    };                                  // <--- end of 'for' (of rule \rho3)
```

```
// ====================== checking termination S(n_0) == E(n_0)
                                                 // comment: mArray is eArray
     termination = equal(node[initNode].sArray, node[initNode].mArray);

     } while (!termination);// <--- end of do-while for \rho2 and \rho3

// ========================================================================
//  Printing the spanning tree:

    boolean emptyTree = true;
    for (int i=0; i<N; i++)
        { emptyTree = emptyTree && (node[initNode].sArray[i] == 0); };
    if (emptyTree)
          { System.out.print("The spanning tree starting from " +
                "node " + initNode + " is node "+ initNode +" only.\n");}
    else { System.out.print("The arcs of a spanning tree, starting " +
                            "from node " + initNode + "\nand reaching "+
                            "all the reachable nodes, are: \n");
           treePrint(node, initNode); };
  }                                                    // <--- end of main
}                                    // <--- DistributedSpanningTree class

/**
 * input:
 * ------------------------------------------------------------------------
 * javac DistributedSpanningTree.java
 * java  DistributedSpanningTree 5 0
 *
 * output:   (with traceon == false)
 * --------------------------------
 * The random undirected graph has 5 node(s) (from node 0 to node 4).
 * The symmetric adjacency matrix is:
 * 0 :  0 0 0 1 1
 * 1 :  0 0 0 1 0
 * 2 :  0 0 0 0 1
 * 3 :  1 1 0 0 0
 * 4 :  1 0 1 0 0
 *
 * Every node is connected to the other nodes with probability 40 %.
 * The initial node is: 0
 *
 * The arcs of a spanning tree, starting from node 0
 * and reaching all the reachable nodes, are:
 * 0 -- 3
 * 3 -- 1
 * 0 -- 4
 * 4 -- 2
 * ------------------------------------------------------------------------
 * ------------------------------------------------------------------------
 * javac DistributedSpanningTree.java
 * java  DistributedSpanningTree 4 0
 *
 * output:   (with traceon == true)
 * --------------------------------
 * The random undirected graph has 4 node(s) (from node 0 to node 3).
 * The symmetric adjacency matrix is:
```

```
* 0 :  0 0 1 1
* 1 :  0 0 0 1
* 2 :  1 0 0 0
* 3 :  1 1 0 0
*
* Every node is connected to the other nodes with probability 40 %.
* The initial node is: 0
*
*  ---- Applying \rho1 to node 0
*
* predecessors of node 0:  0 0 0 0
* mArray of node 0:        0 0 0 0
* successors of node 0:    0 0 1 1
*
* predecessors of node 1:  0 0 0 1
* mArray of node 1:        0 0 0 0
* successors of node 1:    0 0 0 1
*
* predecessors of node 2:  1 0 0 0
* mArray of node 2:        1 0 0 0
* successors of node 2:    0 0 0 0
*
* predecessors of node 3:  1 1 0 0
* mArray of node 3:        1 0 0 0
* successors of node 3:    0 1 0 0
*
*  ---- Applying \rho2 to node 2
*
* predecessors of node 0:  0 0 0 0
* mArray of node 0:        0 0 0 0
* successors of node 0:    0 0 1 1
*
* predecessors of node 1:  0 0 0 1
* mArray of node 1:        0 0 0 0
* successors of node 1:    0 0 0 1
*
* predecessors of node 2:  1 0 0 0
* mArray of node 2:        0 0 0 0
* successors of node 2:    0 0 0 0
*
* predecessors of node 3:  1 1 0 0
* mArray of node 3:        1 0 0 0
* successors of node 3:    0 1 0 0
*
* ---- Applying \rho3 to node 2
*
* predecessors of node 0:  0 0 0 0
* mArray of node 0:        0 0 1 0
* successors of node 0:    0 0 1 1
*
* predecessors of node 1:  0 0 0 1
* mArray of node 1:        0 0 0 0
* successors of node 1:    0 0 0 1
*
* predecessors of node 2:  1 0 0 0
* mArray of node 2:        0 0 0 0
```

```
* successors of node 2:    0 0 0 0
*
* predecessors of node 3:  1 1 0 0
* mArray of node 3:        1 0 0 0
* successors of node 3:    0 1 0 0
*
* ---- Applying \rho2 to node 3
*
* predecessors of node 0:  0 0 0 0
* mArray of node 0:        0 0 1 0
* successors of node 0:    0 0 1 1
*
* predecessors of node 1:  0 0 0 1
* mArray of node 1:        0 0 0 1
* successors of node 1:    0 0 0 0
*
* predecessors of node 2:  1 0 0 0
* mArray of node 2:        0 0 0 0
* successors of node 2:    0 0 0 0
*
* predecessors of node 3:  1 0 0 0
* mArray of node 3:        0 0 0 0
* successors of node 3:    0 1 0 0
*
*  ---- Applying \rho2 to node 1
*
* predecessors of node 0:  0 0 0 0
* mArray of node 0:        0 0 1 0
* successors of node 0:    0 0 1 1
*
* predecessors of node 1:  0 0 0 1
* mArray of node 1:        0 0 0 0
* successors of node 1:    0 0 0 0
*
* predecessors of node 2:  1 0 0 0
* mArray of node 2:        0 0 0 0
* successors of node 2:    0 0 0 0
*
* predecessors of node 3:  1 0 0 0
* mArray of node 3:        0 0 0 0
* successors of node 3:    0 1 0 0
*
*  ---- Applying \rho3 to node 1
*
* predecessors of node 0:  0 0 0 0
* mArray of node 0:        0 0 1 0
* successors of node 0:    0 0 1 1
*
* predecessors of node 1:  0 0 0 1
* mArray of node 1:        0 0 0 0
* successors of node 1:    0 0 0 0
*
* predecessors of node 2:  1 0 0 0
* mArray of node 2:        0 0 0 0
* successors of node 2:    0 0 0 0
*
```

```
 * predecessors of node 3:  1 0 0 0
 * mArray of node 3:        0 1 0 0
 * successors of node 3:    0 1 0 0
 *
 *  ---- Applying \rho3 to node 3
 *
 * predecessors of node 0:  0 0 0 0
 * mArray of node 0:        0 0 1 1
 * successors of node 0:    0 0 1 1
 *
 * predecessors of node 1:  0 0 0 1
 * mArray of node 1:        0 0 0 0
 * successors of node 1:    0 0 0 0
 *
 * predecessors of node 2:  1 0 0 0
 * mArray of node 2:        0 0 0 0
 * successors of node 2:    0 0 0 0
 *
 * predecessors of node 3:  1 0 0 0
 * mArray of node 3:        0 1 0 0
 * successors of node 3:    0 1 0 0
 *
 * The arcs of a spanning tree, starting from node 0
 * and reaching all the reachable nodes, are:
 * 0 -- 2
 * 0 -- 3
 * 3 -- 1
 * ------------------------------------------------------------------
 */

/**
 * ==================================================================
 *                          The ZeroOneGraph class
 *
 * This class ZeroOneGraph defines a undirected graph with N (> 0) nodes.
 * ==================================================================
 */
import java.util.Random;

public class ZeroOneGraph {
   final  int N;              // N = number of nodes of the graph. N > 0.
   static int [][] graph;     // an N x N array of 0's and 1's.

// -------------------- beginning of constructor ----------------------
   public ZeroOneGraph(int N, int PerCentPROB) {
   /** Generation of a random, undirected graph with N (> 0) nodes.
    *  The random variable randomConnect controls the connectivity:
    *      node i is connected with node j, for i != j, with percent
    *      probability PerCentPROB:  0 <= PerCentPROB <= 100
    */
      this.N               = N;
      this.graph           = new int [N][N];
      Random randomConnect = new Random();   // random graph connectivity

    // graph[i][j] = 1 iff there is an arc from node i to node j
    // graph[i][j] = 0 iff there is no arc from node i to node j
```

```
      for(int i=0;i<N;i++) {
        for(int j=i+1;j<N;j++) {
            if (randomConnect.nextInt(100) < PerCentPROB)
                 { graph[i][j] = 1; graph[j][i] = 1; } // symmetric graph
            else { graph[i][j] = 0; graph[j][i] = 0; };// symmetric graph
        }
      }
   }
// --------------------- end of constructor ----------------------------

// --------------------- printing the graph ----------------------------
   public synchronized void graphPrint () {
     System.out.println("The random undirected graph has "+N+" node(s) "+
                        "(from node 0 to node " + (N-1) + ").\n" +
                        "The symmetric adjacency matrix is:");
     for(int i=0;i<N;i++) {
        System.out.print(i + " :  ");
        for(int j=0;j<N;j++) {
            System.out.print( graph[i][j] + " " );
        }; System.out.println();
      }; System.out.println();
   }
}
// ------------------------------------------------------------------------
```

Index

abort operation, 101, 104
abstract view of a transaction, 104
action, 85
Alternating Bit protocol, 96
ancestor node, 9
arithmetic expressions:
- operational semantics, 3
atomic sequence of statements, 44
atomicity, 44
atomicity of operations, 17
atomicity of sequences of statements, 17
atomicity of statements, 17, 43

Bakery protocol, 42
bisimilarity, 86
bisimulation congruence, 86
bisimulation equivalence, 86
boolean expressions:
- operational semantics, 3
bounded buffer, 34
busy waiting, 18, 28, 33

CCS calculus: pure calculus, 85
CCS calculus: value passing calculus, 94
CCS context, 87
CCS term, 85
circular buffer, 20, 34
command for semaphores: *signal*(s), 17
command for semaphores: *wait*(s), 17
command: multiple **fork-join**, 8
command: single **fork-join**, 7
command: **await**, 42
command: **await**
- in Peterson's algorithm, 42, 48
command: **cobegin-coend**, 9
command: **do-od**, 5
command: **if-fi**, 5
command: **synchronized**, 70
command: **try-catch**, 66
command: *notifyAll*()
- for resuming threads, 70
command: *notify*()
- for resuming a thread, 70
command: *sleep*(n)
- for suspending a thread, 69, 76
command: *wait*()
- for stopping a thread, 70
command: *join*()
- for joining threads, 66
command: *start*()
- for running a thread, 64
commit operation, 101, 104
ConcurQueueTester.class, 83
condition queue, 70, 71
condition variable, 32
conditional critical region, 28
conflicting operations, 107
consumer, 24
consumer process, 20
control flow, 8
control flow multigraph, 8
cooperation: bad, 38
cooperation: good, 38
critical region, 26
critical regions in Java, 77
critical section, 19, 42, 45

database, 101
datum, 101
deadlock, 24, 114
deadlock in critical regions, 27
deadlock in the distributed computation of spanning trees, 53
deadlock:
- absence in Peterson's algorithm, 45, 47, 70, 90
descendant node, 9
detecting termination: Topor's rules, 59
deterministic commands:
- operational semantics, 4
deterministic computations, 3

dining philosophers problem, 36

end-token ▲, 51
entry procedures, 32
entry protocol, 111
Euclid's algorithm, 4
exit protocol, 111
Expansion Theorem
- for ≈ in pure CCS, 88
Expansion Theorem
- for ≈ in value-passing CCS, 96
Expansion Theorem
- for = in pure CCS, 89

failure in Peterson's algorithm, 44
Fair Finite Delay hypothesis, 53, 62
fairness assumption, 100
Finite Delay hypothesis, 54, 117
first-in-first-out policy, 39
five philosophers problem, 36

get operation, 23, 24
guarded commands, 5
guarded commands:
- operational semantics, 6

history, 107
history: equivalence , 107
history: serializable, 108, 113

identifier, 85
instruction: **new**, 64
interface: EntryExitProtocol, 67
interface: Monitor
(for the forks of the five philosophers), 80
interleaving, 88

Java interface, 65

label, 85
Law (or Theorem) of Expansion
- for ≈ in pure CCS, 88
Law (or Theorem) of Expansion
- for ≈ in value-passing CCS, 96
Law (or Theorem) of Expansion
- for = in pure CCS, 89
laws for τ, 87, 88, 96
laws for monoid, 87, 88, 96
laws for renaming, 87, 89
laws for restriction, 87, 89
linearization of an execution of a distributed algorithm, 52
location, 101
lock for Java synchronization, 70
lock for reading, 109
lock for writing, 110
lock-free synchronization, 63
locking operation: conflict, 111
locking protocol, 109
locking protocol:
- strict two phase, 113
locking protocol:
- strict two phase scheduler, 113
locking protocol: two phase, 112
locking protocol: two phase,
- growing phase, 112
locking protocol: two phase,
- shrinking phase, 112
lost update problem, 102

mark-token ⊗, 51
matrix multiplication, 11
monitor, 31
monitor: condition, 31
monitor: Java concept, 70
monitor: release procedure, 31
monitor: take procedure, 31
multiple inheritance, 65
mutex queue, 70, 71
mutual exclusion, 115
mutual exclusion for critical sections:
- Java realization using binary semaphores, 77
mutual exclusion:
- in Peterson's algorithm, 42, 45, 47, 70, 90

name, 85
non-critical section, 42, 45
nondeterministic commands:
- operational semantics, 5
nondeterministic computations, 5
nondeterministic Euclid's algorithm, 6

obtaining a lock, 109
operation in conflict with another, 107
order on a transaction, 104
overtaking:
- bounded in Peterson's algorithm, 45, 48, 70, 92
overtaking:
- bounded of degree k in Peterson's algorithm, 45

Peterson's algorithm for n processes, 49
Peterson's algorithm for 3 processes, 50
Peterson's algorithm: n processes, 49
Peterson's algorithm: two processes, 42, 48
prefix sums, 12
prefix sums:
- time × processors complexity, 17
prefix sums: improved rules, 15
prefix sums: rules, 14
prefix sums: tree of processors, 13
private semaphores, 22
procedures, 32
process, 64, 85
process call, 7
process declaration , 7
process failure:
- in Peterson's algorithm, 44
process starvation:
- in Peterson's algorithm, 44
process: dynamic, 7
process: finite, 88
process: operational semantics, 85
process: stable, 88
process: static, 7
producer process, 20
program: BinarySemaphore, 77
program: BoudedBufferMonitor, 73
program: BoundedBuffer, 78
program: BoundedBufferMonitorTester, 75
program: ConcurQueueMonitorTester, 83
program: Consumer, 75
program: Counter, 65
program: CountingSemaphore, 76
program: Fibonacci, 66
program: ForkMonitor, 80
program: ForkMonitorTester, 81
program: MerryChristmas, 64
program: PetersonProtocolTester
- for testing Mutual Exclusion, 69
program: PetersonTwoProcesses, 68
program: PhilosopherThread, 81
program: Producer, 75
program: ProtocolThread, 68
program: QueueMonitor, 82
program: QueueUser, 83
program: RunnableCounter, 65
put operation, 22

QueueMonitor.class, 83
QueueMonitor$1.class, 83
QueueMonitor$Element.class, 83
QueueUser.class, 83

read operation, 104
readset, 115
receive operation, 21
recovery manager, 103
recurrence relations: Fibonacci, 10
releasing a lock, 109
repeat wave, 59
repeats •, 58
run():
- overriding it for constructing a thread, 64
Runnable:
- implementing it for constructing a thread, 65

scheduler, 103
scheduler: locking protocol
- conservative two phase, 115
scheduler: locking protocol
- two phase, 112
semaphore, 17
semaphore: $P(s)$ operation, 17
semaphore: $V(s)$ operation, 17
semaphore: binary, 77
semaphore: boolean, 18
semaphore: counting, 17, 18, 76
semaphore: mutual exclusion, 19
semaphore: parametrized wait operation, 35
semaphore: realized by critical regions, 29

semaphore: room, 39
semaphore: *signal*(s) operation, 17
semaphore: *wait*(s) operation, 17
send operation, 21
serializability theorem, 108
Serializability Theory, 103
serialization graph, 108
serialization graph: acyclic, 108
spanning tree, 51
starvation, 18, 20, 22, 37
starvation in Peterson's algorithm, 44
starvation:
- absence in Peterson's algorithm, 45, 47
state of a thread:
- Enabled, 71
state of a thread:
- Locking the object o, 70
state of a thread:
- Waiting for object o, 70

Termination Condition for distributed computation of spanning trees, 53
termination detection, 58
termination detection algorithm:
- complexity, 62

test-and-set operation, 19
thread, 64
Thread:
- extending it for constructing a thread, 64
token wave, 59
token, black ▲, 59
token, white △, 59
tokens, 58
transaction, 101, 104
transaction concatenation, 108
transaction manager, 103
trying protocol, 111

unlock after reading, 110
unlock after writing, 110

value, 101
victim transaction, 114

wait-for graph, 114
wait-free synchronization, 63
waiting section, 45
write operation, 104
writeset, 115

References

1. P. Ancilotti and M. Boari. *Principles and Techniques of Concurrent Programming.* UTET Libreria, 1987. (In Italian).
2. M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall, 1990.
3. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
4. P. Brinch-Hansen. Structured multiprogramming. *CACM*, 15(7), 1973.
5. P. Brinch-Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE-1(2), 1975.
6. R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics-based Verification Tool for Finite-state Systems. In *Proceedings of the Workshop on Automated Verification Methods for Finite-state Systems*, Lecture Notes in Computer Science 407. Springer-Verlag, 1989.
7. M. Conway. A multiprocessor system design. In *Proceedings AFIPS Fall Joint Computer Conference*, volume 14. Spartan Books, Las Vegas, 1963.
8. E.W. Dijkstra. The structure of the THE multiprogramming system. *CACM*, 8(5), 1968.
9. E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewod Cliffs, N.J., 1976.
10. Vijay K. Garg. *Concurrent and Distributed Computing in Java.* Wiley & Sons, 2004.
11. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Specification Language.* Addison-Wesley, Second edition, 2000.
12. C.A.R. Hoare. Towards a theory of parallel programming. In C.A.R. Hoare and Perrot, editors, *Operating Systems Techniques*, pages 61–71. Academic Press, New York, 1972.
13. C.A.R. Hoare. Monitors: An operating system structuring concept. *CACM*, 17(10):549–557, October 1974.
14. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.
15. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.
16. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
17. Department of Defense. *Reference Manual for the Ada Programming Language.* USA, Department of Defense, January 1983.
18. G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.
19. P. Sestoft. *Java Precisely.* The MIT Press, 2002.
20. R. W. Topor. Termination detection for distributed computations. *Information Processing Letters*, 18(1):33–36, 1984.
21. D. Walker. Analysing mutual exclusion algorithms using CCS. Technical Report ECS-LFCS-88-45, LFCS Edinburgh University, Edinburgh (Scotland), 1988.
22. G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* The MIT Press, Cambridge, Massachusetts, 1993.

23. N. Wirth. Modula: a programming language for modular multiprogramming. *Software Practice and Experience*, 7(1), 1977.
24. N. Wirth. *Programming in Modula-2*. Springer Verlag, 1982.