Appendix A. The Pseudocode Notation

The algorithms are written in a pseudocode notation that is intended to be easy to translate into a programming language. It not only allows a language-independent presentation of the models and algorithms, but also enables us to abbreviate computations that are trivial for a programmer to supply. Examples are:

```
number[i] ← 1 + max(number)
```

in the bakery algorithm (Algorithm 5.2), and

```
for all other nodes
```

in the Byzantine Generals algorithm (Algorithm 12.2).

Structure

The title area gives the name and reference number of the algorithm. The following area is used for the declaration of global variables. Then come the statements of the processes. For two processes, which are named `p` and `q`, the statements are written in two columns. Algorithms for *N* processes are written in a single column. The code for each process is identical except for the ID of each process, written `i` in the concurrent algorithms and `myID` in the distributed algorithms.

In the distributed algorithms, the algorithm is given for each *node*. The node itself may contain several processes which are denoted by giving them titles; see Algorithm 11.3 for an example.

Monitors and other global subprograms that can be executed by any

Syntax

Text from the symbol `//` until the end of a line is a comment.

Data declarations are of the form:

```
type-name variable-name ← initial value
```

The most common type is `integer`, which represents any integer type; implementations can use other types such as **byte** or **short**. In some algorithms, like producer–consumer algorithms, a data type for arbitrary elements is needed, but its precise specification is not important. In these cases, a name ending in `Type` is used, usually `dataType`.

Arrays are declared by giving the element type, the index range and the initial values:

```
integer array [1.. n] number ← [0,. . . .0]
```

Elementary abstract data types like queues and sets are used as needed.

The arrow `←` is used in assignments, so that we don’t have to choose between languages that use `=` and those that use `:=`. In expressions, mathematical symbols like =, ≠ and ≤ are used.

NOTE

Indentation indicates the substatements of compound statements.

`if` statements are conventional. We are quite flexible in the syntax of loop statements, from conventional `while` statements to abbreviations like `do two times`. `loop forever` is used in most concurrent algorithms to denote the indefinite looping of a process.

Semantics

Numbered statements represent *atomic operations* as in Promela; if a continuation line is needed, it is not numbered.

Assignment statements and expression evaluations are atomic statements. The atomicity of evaluating the expression in a condition includes the decision to take one branch or another; for example, in the statement `if a<b then s1 else s2`, if the value of `a` is in fact less than the value of `b`, evaluating `a<b` will cause the control pointer of the process to point to `s1`.

Note that `loop forever`, `else` and `goto` are not numbered; they serve only to specify the control flow and do not denote executable statements.

Synchronization Constructs

The statement `await boolean-valued-expression` is an implementation-independent notation for a statement that waits until the expression becomes true. This can be implemented (albeit inefficiently) by a busy-wait loop that does nothing until the condition is true. Note that in Promela, an `await` statement can be implemented by simply writing an expression:

```
turn == 1
```

The notation `ch` $\Leftarrow$ `x` indicates that the value of `x` is sent on the channel `ch`, and similarly, the notation `ch` $\Rightarrow$ `y` means that the value of the message received from the channel `ch` is assigned to the variable `y`. In CSP and Promela these are denoted `ch!x` and `ch?y`, but the use of arrows clarifies the data flow.

PREV
13. Real-Time Systems

NEXT
B. Review of Mathematical Logic