Carnegie Mellon University
Software Engineering Institute

Concepts of Concurrent Programming

David W. Bustard
University of Ulster

April 1990

Preface

Capsule Description

A concurrent program is one defining actions that may be performed simultaneously. This module discusses the nature of such programs and provides an overview of the means by which they may be constructed and executed. Emphasis is given to the terminology used in this field and the underlying concepts involved.

Scope

This module is largely concerned with *explicit concurrency*, where concurrent behavior is specified by the program designer. Particular attention is paid to programs that can be considered *inherently concurrent*, that is, programs that are constructed to control or model physical systems that involve parallel activity. The module also includes a brief introduction to *performance-oriented* concurrency, where concurrency is used to improve program performance by taking advantage of hardware support for parallel processing.

The module is divided into three sections. The first deals with basic concepts in concurrent programming, covering characteristic attributes, formal properties, standard design problems, and execution details. The second section discusses the steps in constructing concurrent programs from specification to coding. The final section briefly examines concurrency from the point of view of some common application areas.

The module gives a foundation for a deeper study of specific topics in concurrent programming. It also provides the preparatory material for a study of the concurrent aspects of application areas such as real-time (embedded) systems, database systems, operating systems, and many simulation systems.

Philosophy

A sequential program is really just a concurrent program in which a single activity is defined. This is not a fanciful idea. In practice, most software design techniques yield program structures that are naturally

concurrent, and developers need to go to some lengths to convert such designs into a sequential form (for an example see [Sutcliffe88]).

Traditionally, this avoidance of a concurrent program representation has occurred for two main reasons. One is the lack of a suitable implementation language for the application concerned; the other is a belief that the concurrency concept is too difficult for the average programmer. In the experience of practitioners [Brinch Hansen77, Gelernter88], the latter argument is unfounded. Language support has indeed been a problem but is one that is diminishing rapidly for most application areas [Wegner89]. At the current rate of progress it seems likely that most programming languages will support the representation of concurrency by the turn of the century. In that event, a study of the concepts of concurrent programming will become an essential first step in understanding programming in general. This module is a contribution to that vision.

Acknowledgements

I am very grateful to Gary Ford for his guidance and encouragement through every stage in the production of this module. Linda Pesante also earns my sincere thanks for her infectious enthusiasm and her ability to turn apparently neat phrases into much neater ones. I am also indebted to Karola Fuchs, Sheila Rosenthal, and Andy Shimp, who provided an excellent library service that managed to be both efficient and friendly.

The technical content of the module has benefited significantly from suggestions made by Mark Ardis, Lionel Deimel, and Robert Firth. I am also grateful to Daniel Berry, who gathered some initial material.

Finally, I would like to thank Norm Gibbs for the opportunity to produce this module and for his concern that the experience should be enjoyable. It was!

Author's Address

Comments on this module are solicited, and may be sent to the SEI Software Engineering Curriculum Project or to the author:

David W. Bustard
Department of Computing Science
University of Ulster
Coleraine BT52 1SA
Northern Ireland

Concepts of Concurrent Programming

Outline

1. Basic Concepts

- 1.1. The Nature of Concurrent Programs
  - 1.1.1. Implicit and explicit concurrency
  - 1.1.2. Processes and concurrent programs: basic definitions
  - 1.1.3. Distinguishing concurrent, parallel, and distributed programs
  - 1.1.4. Distinguishing concurrent programs and concurrent systems
  - 1.1.5. Nondeterminism
  - 1.1.6. Process interaction
- 1.2. Problems in Concurrent Programs
  - 1.2.1. Violating mutual exclusion
  - 1.2.2. Deadlock
  - 1.2.3. Indefinite postponement (or starvation or lockout)
  - 1.2.4. Unfairness
  - 1.2.5. Busy waiting
  - 1.2.6. Transient errors
- 1.3. Properties of Concurrent Programs
  - 1.3.1. Safety
  - 1.3.2. Liveness
- 1.4. Executing Concurrent Programs
  - 1.4.1. Measures of concurrency
  - 1.4.2. Execution environments
  - 1.4.3. Patterns of execution
  - 1.4.4. Process states
  - 1.4.5. Process scheduling

Annotated Outline

1. Basic Concepts

This section provides introductory definitions and discussion of the main concepts and terms used in concurrent programming. Further explanations, with illustrations, may be found in any basic text in this area [Ben-Ari82, Bustard88, Schiper89, Whiddett87]. Concurrency concepts are also covered in most books on operating systems [Deitel84, Habermann76, Lister85] and in texts addressing the concurrent aspects of specific programming languages [Burns85, Gehani84, Gehani85, Holt83]. An introduction to distributed and parallel programming may be found in [Critchlow88, Perrott87].

1.1. The Nature of Concurrent Programs

1.1.1. Implicit and explicit concurrency

In principle, most programs may be considered concurrent in that they are likely to:

- contain independent processing steps (at the block, statement, or expression level) that may be executed in parallel; or
- trigger device operations that may proceed in parallel with the execution of the program.

This may be termed *implicit concurrency*. *Explicit concurrency* is where concurrent behavior is specified by the program designer.

An introduction to the detection of implicit concurrency in a program may be found in [Perrott87]. A more detailed discussion is presented in [Polychronopoulous88].

Generally, the detection of concurrency implies the identification of sequences of independent array or arithmetic operations that might be executed in parallel (e.g. setting array elements to zero). Such analysis is carried out to improve the performance of existing sequential code. Typical improvements are, however, only a fraction of the potential speed-up that might be achieved by restructuring the programs involved.

The remainder of this module is concerned with explicit concurrency.

1.1.2. Processes and concurrent programs: basic definitions

A *sequential program* specifies sequential execution of a list of statements; its execution is called a *process*. A *concurrent program* specifies two or more sequential programs that may be executed concurrently as *parallel processes* [Andrews83]. In many languages, *process* is also the name of the construct used to describe process behavior [Bustard88, Gehani89, Holt83]; one notable exception is Ada, which uses the name *task* for this purpose [Burns85].

1.1.3. Distinguishing concurrent, parallel, and distributed programs

In the literature, a concurrent program is commonly discussed in the same context as parallel or distributed programs. Unfortunately, few authors give precise meanings to these terms and the meanings that are offered tend to conflict. On balance, the following definitions seem appropriate:

- A *concurrent program* defines actions that may be performed simultaneously.
- A *parallel program* is a concurrent program that is designed for execution on parallel hardware.
- A *distributed program* is a parallel program designed for execution on a network of autonomous processors that do not share main memory [Bal89].

Thus, *concurrent program* is a generic term used to describe any program involving actual or potential parallel behavior; *parallel* and *distributed programs* are sub-classes of concurrent program that are designed for execution in specific parallel processing environments.

Where it is known that a single processor will be used to execute a program expressed in a concurrent form, the concurrency involved is sometimes referred to as *pseudoparallelism* [Bal89, Schiper89]. A *quasiparallel* program [Schiper89] is a pseudoparallel program in which processes execute cooperatively by transferring control to each other using a *coroutine* mechanism. (This is the basis of the concurrency model supported in the programming language Modula-2 [Ford86].)

1.1.4. Distinguishing concurrent programs and concurrent systems

A concurrent program is primarily a coherent unit of software. If two pieces of communicating software run concurrently, the result is a concurrent program when the two pieces form a conceptual whole; otherwise, the situation is viewed as two programs communicating through an agreed protocol. The communicating programs do,

however, constitute a *concurrent system* (or *parallel system* or *distributed system*, as appropriate).

1.1.5. Nondeterminism

A sequential program imposes a total ordering on the actions it specifies. A concurrent program imposes a partial ordering, which means that there is uncertainty over the precise order of occurrence of some events; this property is referred to as *nondeterminism*. A consequence of nondeterminism is that when a concurrent program is executed repeatedly it may take different execution paths even when operating on the same input data.

The classification of a program as deterministic or nondeterministic will depend on which of its actions are considered significant. Most programs are nondeterministic when viewed at a low enough level in their execution but it is generally external behavior that dictates the overall classification.

1.1.6. Process interaction

All concurrent programs involve process interaction. This occurs for two main reasons:

1. Processes compete for exclusive access to shared resources, such as physical devices or data.
2. Processes communicate to exchange data.

In both cases it is necessary for the processes concerned to *synchronize* their execution, either to avoid conflict, when acquiring resources, or to make contact, when exchanging data.

Processes can interact in one of two ways: through shared variables, or by message passing from one to another. Process interaction may be *explicit* within a program description or occur *implicitly* when the program is executed. In particular, there is implicit management of machine resources, such as processor power and memory, that are needed to run a program. (Further details may be found in Section 1.4)

1.1.6.1. Resource management

A process wishing to use a *shared resource* (e.g. a printer) must first *acquire* the resource, that is, obtain permission to access it. When the resource is no longer required, it is *released*; that is, the process relinquishes its right of access.

If a process is unable to acquire a resource, its execution is usually suspended until that resource is available. Resources should be administered so that no process is delayed unduly.

A process may require access to one or more resources simultaneously, and those resources may be of the same or different types. The resources may be defined statically within the program or created and destroyed as the program executes. Also, some resources may be acquired for exclusive use by one process while others may be shared if used in a particular way. For example, several processes may inspect a data item simultaneously but only one process at a time may modify it. See [Bustard88] for a detailed discussion of these possibilities.

1.1.6.2. Communication

Inter-process communication is one of the following:

- *Synchronous*, meaning that processes synchronize to exchange data.
- *Asynchronous*, meaning that a process providing data may leave it for a receiving process without being delayed if the receiving process is unable to take the data immediately; the data is held temporarily in a *communication buffer* (a shared data structure). Where several data items are buffered, these are made available to the receiving process in the order in which they arrive at the buffer.

1.2. Problems in concurrent programs

1.2.1. Violating mutual exclusion

Some operations in a concurrent program may fail to produce the desired effect if they are performed by two or more processes simultaneously. The code that implements such operations constitutes a *critical region* or *critical section*. If one process is in a critical region, all other processes must be excluded until the first process has finished. When constructing any concurrent program, it is essential for software developers to recognize where such *mutual exclusion* is needed and to control it accordingly.

Most discussions of the need for mutual exclusion use the example of two processes attempting to execute a statement of the form:

$$x := x + 1$$

Assuming that *x* has the value *12* initially, the implementation of the statement may result in each process taking a local copy of this value, adding one to it and both returning *13* to *x* (unlucky!).

Mutual exclusion for individual memory references is usually implemented in hardware. Thus, if two processes attempt to write the values 3 and 4, respectively, to the same memory location, one access will always exclude the other in time leaving a value of 3 or 4 and not any other bit pattern.

Techniques for implementing mutual exclusion are discussed in Section 2.2.3.

1.2.2. Deadlock

A process is said to be in a state of *deadlock* if it is waiting for an event that will not occur. Deadlock usually involves several processes and may lead to the termination of the program. A deadlock can occur when processes communicate (e.g., two processes attempt to send messages to each other simultaneously and synchronously) but is a problem more frequently associated with resource management. In this context there are four necessary conditions for a deadlock to exist [Coffman71]:

1. Processes must claim exclusive access to resources.
2. Processes must hold some resources while waiting for others (i.e., acquire resources in a piecemeal fashion).
3. Resources may not be removed from waiting processes (no pre-emption).

4. A circular chain of processes exists in which each process holds one or more resources required by the next process in the chain.

Techniques for avoiding or recovering from deadlock rely on negating at least one of these conditions. One of the best documented (though largely impractical) techniques for avoiding deadlock is Dijkstra's Banker's Algorithm [Dijkstra68]. Dijkstra also posed what has become a classic illustrative example in this field, that of the Dining Philosophers [Dijkstra71].

1.2.3. Indefinite postponement (or starvation or lockout)

A process is said to be *indefinitely postponed* if it is delayed awaiting an event that may not occur. This situation can arise when resource requests are administered using an algorithm that makes no allowance for the waiting time of the processes involved. Systematic techniques for avoiding the problem place competing processes in a priority order such that the longer a process waits the higher its priority becomes. Dealing with processes strictly in their delay order is a simpler solution that is applicable in many circumstances. See [Bustard88] for a discussion of these techniques.

1.2.4. Unfairness

It is generally (but not universally) believed that where competition exists among processes of equal status in a concurrent program, some attempt should be made to ensure that the processes concerned make even progress; that is, to ensure that there is no obvious *unfairness* when meeting the needs of those processes. Fairness in a concurrent system can be considered at both the design and system implementation levels. For the designer, it is simply a guideline to observe when developing a program; any neglect of fairness may lead to indefinite postponement, leaving the program incorrect.

For a system implementor it is again a guideline. Most concurrent programming languages do not address fairness. Instead, the issue is left in the hands of the compiler writers and the developers of the run-time support software.

Generally, when the same choice of action is offered repeatedly in a concurrent program it must not be possible for any particular action to be ignored indefinitely. This is a weak condition for fairness. A stronger condition is that when an open choice of action is offered, any selection should be equally likely.

1.2.5. Busy waiting

Regardless of the environment in which a concurrent program is executed, it is rarely acceptable for any of its processes to execute a loop awaiting a change of program state. This is known as *busy waiting*. The state variables involved constitute a *spin lock*. It is not in itself an error but it wastes processor power, which in turn may lead to the violation of a performance requirement. Ideally, the execution of the process concerned should be suspended and continued only when the condition for it to make progress is satisfied.

1.2.6. Transient errors

In the presence of nondeterminism, faults in a concurrent program may appear as *transient errors*; that is, the error may or may not occur depending on the execution path taken in a particular activation of the

program. The cause of a transient error tends to be difficult to identify because the events that precede it are often not known precisely and the source of the error cannot, in general, be found by experimentation. Thus, one of the skills in designing any concurrent program is an ability to express it in a form that guarantees correct program behavior despite any uncertainty over the order in which some individual operations are performed. That is, there should be no part of the program whose correct behavior is *time-dependent*.

1.3. Properties of Concurrent Programs

The requirements for a concurrent program can be defined in terms of *properties* that it must possess. If the properties are expressed formally (mathematically), then it may be possible to verify formally that an implementation has these properties. Many properties can be classified as either a *safety* or a *liveness* property [Lamport89].

1.3.1. Safety

Safety properties assert what a program is allowed to do, or equivalently, what it may not do. Examples include:

- Mutual exclusion: no more than one process is ever present in a critical region.
- No deadlock: no process is ever delayed awaiting an event that cannot occur.
- Partial correctness: if a program terminates, the output is what is required.

A safety property is expressed as an *invariant* of a computation; this is a condition that is true at all points in the execution of a program. Safety properties are proved by *induction*. That is, the invariant is shown to hold true for the initial state of the computation and for every transition between states of the computation.

1.3.2. Liveness

Liveness (or *progress* [Chandy88]) properties assert what a program must do; they state what will happen (eventually) in a computation. Examples include:

- Fairness (weak): a process that can execute will be executed.
- Reliable communication: a message sent by one process to another will be received.
- Total correctness: a program terminates and the output is what is required.

Liveness properties are expressed as a set of liveness *axioms*, and the properties are proved by verifying these axioms. Safety properties can be proved separately from liveness properties, but proofs of liveness generally build on safety proofs.

1.4. Executing Concurrent Programs

1.4.1. Measures of concurrency

Concurrent behavior can be measured in several ways. In practice, the measures are merely rough classifications of behavior that help characterize a program. These measures are given names here, for convenience, but there is no consensus on naming. In particular,

references to the term *grain* or *granularity* of concurrency in the literature may mean any of the following measures:

- The *unit of concurrency* is the language component on which process behavior is defined [Bal88]. It may be an element in an expression; it may be a program statement; but most commonly it is a program block.
- The *level of concurrency* is the mean number of active processes present during the execution of a program.
- The *scale of concurrency* is the mean duration (or lifetime) of processes in the execution of a program [Bustard88]; there is an overhead in initiating a concurrent activity, and so ideally its duration should be sufficiently long to make that overhead negligible.
- The *grain of concurrency* is the mean computation time between communications in the execution of a program [Bal89]; this should be relatively large if a physical distribution of processes is required.

1.4.2. Execution environments

Programs involving large-scale concurrent behavior (comprising processes of relatively long duration) are executed most commonly on a single processor computer in which the processor is shared among the active processes. This is known as *multiprogramming* (or *multitasking*). *Multiprocessing* occurs on a *multiprocessor*, a computer in which several (usually identical) processors share a common primary memory.

Multicomputers use separate primary memory, and their execution of processes is known as *distributed processing*. *Closely coupled* multicomputers have fast and reliable point-to-point interprocessor links; *loosely coupled* systems communicate over a network that is much slower and much less reliable. Components of a multicomputer may be in the same vicinity or physically remote from each other. In [Bal89] these are referred to as *workstation-LANs* (Local Area Networks) and *workstation-WANs* (Wide Area Networks), respectively.

Small-scale concurrent programs are usually executed by *array* or *vector processor* computers that apply the same operation to a number of data items at the same time. This is known as *synchronous processing* [Perrott87]. *Dataflow* and *reduction* machines apply different operations to different data items simultaneously [Treleaven82]. These latter machines are still largely experimental. A detailed presentation of the hardware available for parallel processing is given in [Hwang84]. A collection of early papers on parallel processing may be found in [Kuhn81].

1.4.3. Patterns of execution

Most commonly, a concurrent program starts as a single process and subdivides into multiple processes at some point in its execution. The spawned processes may be activated individually or in sets. The processes thus activated may be able to subdivide in the same way. There are two main models of execution:

1. The spawned processes, when activated, execute independently of the process that triggers their execution.

2. The triggering process *forks* into multiple processes which, when complete, *join* to form a single process again.

Most programming languages support the *fork-and-join* model.

1.4.4. Process states

A process exists in one of three states (there is no agreement on the names used):

1. *Awake*, meaning that the process is able to execute.

- *Asleep* (or *blocked*), meaning that the process is suspended awaiting a particular event (e.g., message arrival or resource available).
- *Terminated*, meaning that the execution of the process has finished.

Processes that are *awake* can be further divided into those that are *running* (executing) and those that are *ready* to run as soon as a processor becomes available.

1.4.5. Process scheduling

In exceptional circumstances, a concurrent program may run directly on bare hardware. More usually, however, it will execute on top of support software that provides a more abstract interface to that hardware. This is known as the system *kernel* or *nucleus*. One component of the nucleus is the *scheduler*, which is responsible for the allocation of processors to processes, that is, the resolution of the mismatch between the number of processes that can execute and the number of processors available to execute them. In distributed systems, the scheduler itself may be distributed [Bamberger89].

The processes in some concurrent programs are assigned explicitly to particular processors by the program designer. More commonly, however, the mapping is handled implicitly by the scheduler.

Processes often execute with different priorities. One objective of the scheduler is to ensure that all running processes have no lower a priority than those that are in a ready state. Priorities may be assigned explicitly by the program designer or be set and adjusted implicitly by the scheduler.

The compilation of a concurrent program results in the generation of calls to the kernel that may trigger scheduling operations. Any entry to the kernel provides an opportunity to suspend the process involved and select another for execution. In some cases, normal program behavior may result in an acceptably even distribution of processor power over the competing processes. However, when processing power is scarce it is desirable to implement some form of *time slicing* to ensure that all processes make steady progress. This is often implemented with the assistance of a system clock that interrupts at least one processor at regular intervals.

2. Program Construction

Most aspects of concurrent program construction are covered by other curriculum modules. This section provides an introduction to that material and adds supplementary information where appropriate.

2.1. Development Methods

Concurrent and sequential programs are developed in much the same way [Scacchi87]. There are three main phases involved: requirements analysis, software design, and software coding [Rombach90]. In practice, these phases are not clearly distinguished [Swartout82]. This is particularly evident in the notations used for describing the products of each phase, as often the same notation can be applied in more than one phase [Rombach90]. In what follows, methods are discussed under the headings *requirements*, *design*, or *coding* to indicate their main level of concern; however, it should be understood that the techniques involved can often be used in several contexts.

2.2. A Requirements View

The nature and management of software requirements, in general, are discussed in [Brackett90]. The small part of this material dealing with concurrent systems is elaborated in [Gomma89]. The formal specification of requirements is covered in [Berztiss87].

At the requirements level, properties of programs are emphasized rather than their structure. This section identifies some techniques for describing properties of concurrent systems formally, in the mathematical sense. Note, however, that there are important properties that cannot, at present, be expressed adequately this way. These include performance requirements such as a stipulation of an average response time to an event. Formal descriptions are therefore used in conjunction with natural language statements of requirements as a way of making them more precise.

2.2.1. Temporal logic

Temporal logic can be used for the formal description of either concurrent or sequential programs. It is an extension of classical logic to deal with time [Galton87].

Each execution of a program yields a computation that can be expressed as a linear sequence of states and associated events. This is known as a *trace*. Temporal logic is a formalism for specifying structures of traces. Two varieties of temporal logic are in use, distinguished by the view they take of the possible traces for a given program [Pneuli86]:

- *Linear time temporal logic* considers traces individually.
- *Branching time temporal logic* assumes that the possible traces form a computation tree that retains information about the states at which nondeterministic choices were made.

In either case, a set of operators (and accompanying symbols) are defined and used in the construction of *temporal formulae*. Pneuli [Pneuli86] defines (strong) operators *next*, *until*, *previous* and *since*, on which are built derived operators, such as *eventually*, *henceforth* and *unless*. These in turn are used in formulae to make statements of the form:

- if *p* now then eventually *q*
- every *p* is followed by a *q*

where *p* and *q* are *state formulae* evaluated for particular states of the computation. Temporal logic is used to describe safety and liveness properties of programs.

2.2.2. Petri nets

Petri nets [Peterson81, Reisig85] offer a means of modeling information flow in a concurrent system. There are several varieties of net. In a "condition/event" net, events are linked by conditions. Each event has a set of input and a set of output conditions. Output conditions from one event may serve as input conditions to another, thereby forming the net. An event occurs when all of the necessary input conditions are satisfied. Events may occur simultaneously. Each such event enables its output conditions, which in turn may enable other events. For simple systems, the net can usefully be summarized in a graphical form.

2.2.3. Process models

The behavior of a concurrent program can be described in terms of communicating processes [Brinksma88, Hoare85, Milner89]. Each process is described in terms of the actions or events in which it is involved. There are, in general, three types of event specified:

1. An event internal to a single process.
2. A synchronization (communication) between one process and one or more others.
3. A synchronization between one or more processes and the environment of the system described.

Process descriptions compose events to constrain their order of occurrence. Typically, processes are defined in a recursive fashion. For example, the action of a clock (process) might be described as an infinite sequence of *tick* and *tock* events, thus:

clock: (*tick*; *tock*; clock)

The precise notation and set of operators available for process description vary from one notation to another, but each supports the same basic approach to system specification.

2.2.4. Finite state machines

A *finite state machine* (FSM) may be used to model the behavioral aspects of a process [Davis88]; concurrent systems may be described by interacting FSMs [Harel88].

An FSM is defined by a set of *states* and *transitions* among them. A *state transition diagram* is a graphical representation of an FSM in which nodes represent states and arcs represent state transitions. An FSM may also be described by a *state transition matrix*.

2.2.5. Data flow diagrams

A *data flow diagram* (DFD) is a graph showing data transformations and repositories (as nodes) and the data flowing among them (as connecting arcs). Such a description is inherently concurrent in that it permits parallel data transformation. For more direct use in describing concurrent systems, *event flows* and *control transformations* (representing constraints on events) have been added to the standard DFD notation [Ward85].

2.3. A Design View

A general introduction to program design is presented in [Budgen89] and the design of concurrent systems covered in [Gomaa89]. Surveys of a

number of techniques in this area may be found in [Davis88]. Gomaa's module is primarily concerned with real-time programs but much of the discussion is relevant to concurrent programs in general. He covers five main methods in detail:

1. *Structured Analysis and Design for Real-Time Systems* [Ward85, Yourdon89].
2. *Naval Research Lab Software Cost Reduction Method* [Parnas84].
3. *Object-Oriented Design* [Booch86].
4. *Jackson System Development for Real-Time Systems* [Sutcliffe88].
5 *Design Approach for Real-Time Systems (DARTS)* [Gomaa87].

Each method is discussed under six headings: overview, basic concepts, steps in method, products of design process, assessment of method, and extensions/variations. The first method is the one used most extensively at present [Wood89], but the popularity of the object-oriented approach to program development is increasing rapidly.

The design of parallel programs (those taking advantage of hardware support for parallel processing) is discussed in [Carriero89]. The following approaches are identified:

- *Result parallelism*, where the design is based on the data structure produced by the program. Separate elements of the data structure can be computed simultaneously.
- *Agenda parallelism*, where the design is based on the sequence of steps in a computation. Some steps may be divided into computations that are performed in parallel (fork-and-join concurrency).
- *Specialist parallelism*, where the design is based on components with a clearly identifiable purpose. This is essentially object-oriented design [Booch86]; the components cooperate, in parallel, to achieve the overall purpose of the program.

The basic approach taken in any particular instance is largely dictated by the nature of the application.

2.4. A Coding View

The representation of concurrent programs is discussed in [Feldman90]. Essentially the main concerns are:

- The representation of processes.
- The representation of a mechanism (or mechanisms) for process interaction.

Concurrent behavior may be expressed directly in a programming notation or implemented by system calls. In a programming notation, a process is usually described in a program block and process instances created through declaration or invocation references to that block.

Process interaction is via shared variables or by message passing from one process to another. Programming languages tend to support only one of these approaches, which are discussed separately in 2.4.1 and 2.4.2 below.

2.4.1. Interaction via shared variables

Shared variables are manipulated in a critical region. Mutual exclusion can be implemented without special language features in some circumstances (e.g., see Dekker's algorithm in [Ben-Ari82]) but,

in general, explicit protection is necessary. This protection is implemented by code that encompasses the critical region, thus:

acquire exclusive access to region
critical region
release exclusive access to region

The main ways to provide the protection are now discussed briefly. Further information may be found in [Raynal86].

2.4.1.1. Status variables

Entry to a critical region can be controlled by a *status variable* (memory bit) that indicates whether or not the region is currently occupied by a process. The status variable itself requires exclusive access. This can be handled by an atomic operation that both inspects and sets the variable without interruption. Most computers provide such an instruction.

The status variable mechanism is simple to understand and implement. However, it is insecure and also potentially wasteful because it encourages busy waiting.

2.4.1.2. Semaphores

A semaphore can be regarded as a high-level abstraction for the status variable mechanism described in Section 2.4.1.1. Entry to and exit from a critical region is controlled by P and V operations, respectively. The notation was proposed by Dijkstra [Dijkstra68], and the operations can be read as “wait if necessary” and “signal” (the letters actually represent Dutch words meaning pass and release). Some semaphores are defined to give access to competing processes in arrival order. The original definition, however, does not stipulate an order; and even some appearing recently [BSI89] are defined that way. The less strict definition gives greater flexibility to the implementor but forces the program designer to find other means of managing queues of waiting processes.

Semaphores are the most commonly used mechanism for controlling mutual exclusion. They are, however, insecure and also too restrictive because they cannot be inspected directly [Ben-Ari82].

2.4.1.3. Conditional critical regions

The conditional critical region concept was proposed by Hoare and Brinch Hansen [Andrews83] to improve on the recognized deficiencies of semaphores. Variables in a critical region are defined as a named *resource*. Critical regions are then preceded by the appropriate resource identification and (optionally) a condition to be satisfied before entry can be effected.

This mechanism has only been implemented experimentally and its relevance is largely historical. However, it has been an important influence on modern programming notations.

2.4.1.4. Monitors

The monitor concept was proposed by Brinch Hansen [Brinch Hansen77] and Hoare [Hoare74]. A monitor improves on the conditional critical region construct by combining regions that

use the same resource in one program block. The monitor is less convenient for the program developer, however, because the suspension and reactivation of processes awaiting a program event must be performed explicitly.

A monitor provides a set of procedures through which operations on shared data are performed. The execution of one procedure by any process causes all other processes attempting entry to the monitor to be delayed until the first process has finished or has been suspended. A comparison of the main implementations of monitors in programming languages may be found in [Bustard88].

2.4.2. Interaction by message passing

Process interaction through message passing has proved to be a more popular model of behavior than that based on shared variables. This is the model used most often by those considering formal aspects of program description [Brinksma88, Hoare85, Milner89] and is the model adopted by the major languages supporting the representation of concurrent behavior: Ada [Burns85] and Concurrent C [Gehani89]. It is also a model amenable to implementation in a distributed environment.

Communication may be *synchronous* or *asynchronous* (as discussed in Section 1.1.5.2). In some formal models, communicating processes name each other [Hoare78], but this does not occur in programming languages because it makes the management of a process library difficult. The alternatives are:

- Processes can name a shared *communication channel* [INMOS88].
- Processes can communicate asymmetrically, with only the sending process naming the receiver [Burns85]. For example, in Ada the instigating process invokes an *entry* procedure made available by the receiving process and the receiving process executes a statement to *accept* the call. Several processes invoking the same entry procedure are made to wait in arrival order. The receiving process may accept one from a number of available entry calls and may put preconditions on their acceptance. The basic communication mechanism, allowing exchange of information, is called a *rendezvous*. Concurrent C [Gehani89] also supports the rendezvous concept but in addition permits asynchronous communication in which the sending process is able to proceed without waiting for data supplied to be accepted explicitly.

2.5. Concurrent Program Evaluation

A number of existing curriculum modules deal with the evaluation of programs and their intermediate specification and design products. A general introduction to validation and verification may be found in [Collofello88], with specific details on testing and analysis provided by [Morell89]. Formal verification is covered in [Berztiss88]. Some aspects of validation and verification specific to concurrent systems are given in [Gomaa89]. A survey of concurrent program evaluation techniques, with particular emphasis on debugging, may be found in [McDowell89].

3. Common Applications

This section looks very briefly at the characteristics of concurrent programs in a few commonly occurring application areas. The first three areas are discussed in greater detail in [Bustard88].

3.1. Real-Time Systems

Real-time systems are described in [Gomaa89]. Gomaa identifies six distinguishing characteristics, some or all of which may be present in any particular instance:

1. A real-time program may be a component of a larger hardware/software system; this is known as an *embedded* application.
2. A real-time program typically interacts with an external environment; the interaction may be with humans, but more commonly, it is with equipment of various kinds.
3. All real-time programs have timing constraints, meaning that they *must* perform certain actions within a defined time period.
4. A real-time program will often have to make control decisions based on input data.
5. A real-time program is usually *event driven* (also known as *reactive* [Pneuli86]), that is, it carries out operations in response to the input it receives (including the passage of time indicated by a real-time clock).
6. A real-time program is usually concurrent in order to respond to, or control, events that may occur simultaneously.

It is normal, when designing a real-time program, to dedicate a separate process to the management of each distinct event source, and this largely dictates the concurrent structure of the program. Typically, process interaction is more concerned with the communication of data rather than the management of resources; indeed, to meet timing constraints there should be no significant delay in a process obtaining what it requires. Real-time programs may execute on top of a run-time kernel or on top of a real-time operating system that implements some or all of the low-level device handling functions.

3.2. General-Purpose Operating Systems

A general-purpose operating system manages the resources of a computing facility to provide a service to one or more users [Deitel84, Habermann76, Lister85]. An operating system is a class of real-time system and possesses most of the distinguishing characteristics listed in Section 3.1 (i.e., 2, 3, 5, 6). As with real-time systems, processes are often dedicated to device handling. Processes are also used to manage requests issued by a user; there is generally one process per user that handles basic communication and others that are created (spawned), when necessary, to perform user-initiated operations.

Resource management is the main concern of an operating system, and that is reflected in the literature. Historically, emphasis has been placed on making best use of the resources available because computing equipment was so expensive. This then tended to increase the likelihood of deadlock and livelock in the systems constructed. Now that equipment is cheaper, there is less sharing. Indeed, many functions of an operating

system are often distributed (e.g., dedicated file and print servers) [Critchlow88, Tanenbaum85]. Consequently, deadlock and livelock problems have become less common.

Examples of operating systems expressed in a monitor-based programming language may be found in [Brinch Hansen77, Joseph84].

3.3. Simulation Systems

A good general introduction to the programming of simulation systems may be found in [Kreutzer86]. Discrete event simulation, where the simulation is driven by events occurring in the system being modeled, is the approach that has been given most attention by computer scientists [Birtwistle79].

Real-time and operating systems are often simulated before they are put into live operation. In particular, this is essential when a program is potentially life-threatening such as in the control system of an aircraft, a pacemaker, or even the doors of an elevator. The real-time program may be placed in a simulated environment or modified slightly to map its environment operations onto standard devices and data files [Bustard88].

Simulation programs may also be constructed to model complex real-world situations in order to gain a greater understanding of the systems involved. Simulation programs are often constructed as sequential programs. However, by using a concurrent structure it is usually possible to model the system under investigation more directly. Because simulation programs undergo frequent modification, to experiment with the model they implement, it is desirable that they be easy to understand.

General simulation models may involve resource management and process communication activities that are considerably more complex than those found in real-time or operating systems. The one standard requirement of simulation applications is a need to model time. In effect, simulated time becomes the common synchronization mechanism for all processes of a simulation program [Bustard88].

3.4. Database Systems

Introductions to database systems may be found in [Date86, Ullman82]; these cover the terms and techniques introduced below. In addition, details of the programming of such systems may be found in [Hughes88]. Concurrency issues, including the management of distributed databases, are given particular attention in [Bernstein87].

A *database* is a structured collection of data items. Databases are generally concurrent systems because of the need to permit more than one user to have access to the database simultaneously. A *transaction* is the sets of actions performed on a set of data items, transforming the database from one consistent state to another. The effect of executing several transactions concurrently must be the same as executing them in sequence.

Forcing transactions into a strict sequence by implementing mutual exclusion for the complete database is one way to guarantee consistency. However, to keep user response at an acceptable level, it is usually essential to implement read and write *locks* on smaller data units. The size of the unit defines the *degree of granularity* involved. A transaction will generally require access to several data units simultaneously so some mechanism for dealing with deadlock is required.

Normally, no attempt is made to avoid deadlock. Instead, it is detected and a recovery implemented by aborting one or more locked transactions and undoing any changes made to the database by those transactions (*roll-back*).

Another approach (*optimistic scheduling*), not involving locks, is to first determine the modifications required and then *commit* the changes (*two-phase commit* policy). At that point, the set of read and write operations on the database are compared with those performed by concurrent transactions that have already been committed. If a conflict is detected, the candidate transaction is rejected; otherwise the database is updated.

Glossary

Each definition given in the glossary has a reference to the section(s) in the outline where the term appears. Definitions of other terms and, in some cases, alternative interpretations of the terms defined below may be found in [IEEE87].

agenda parallelism

a design technique for parallel programs that introduces parallelism in the sequence of steps in a computation [2.3].

array processor

a set of identical processing elements synchronized to perform the same instruction simultaneously on different data [1.4.2].

asleep

a process state in which the process is suspended awaiting a particular event [1.4.4].

asynchronous communication

communication among processes that is achieved by the sender of the information leaving it in a buffer for the receiver to collect [1.1.6.2].

awake

a process state in which the process is able to execute [1.4.4].

blocked

a synonym for *asleep*.

branching time temporal logic

a version of temporal logic that treats the computations resulting from the execution of a program as a tree that retains information about the states at which nondeterministic choices were made [2.2.1].

buffer

a shared data structure that supplies data items in the order in which they were inserted [1.1.6.2].

busy waiting

the action of a process that executes a loop awaiting a change of program state [1.2.5].

close coupling

a point-to-point interprocessor linkbetween computers [1.4.2].

communication channel

a logical link established between pairs of processes to enable them to communicate [2.4.2].

concurrent program

(1) a program specifying two or more sequential programs [1.1.2];

(2) a program specifying actions that may be performed simultaneously [1.1.3].

concurrent system

a set of programs communicating through an agreed protocol [1.1.4].

conditional critical region

a language construct used to identify a critical region, the shared variables that it accesses, and any preconditions for entry to the region [2.4.1.3].

control transformation (Ward/Mellor Real-Time Structured Analysis)

a control function that is defined by means of a state transition diagram [2.2.5].

critical region

a section of code that performs an operation that must not be executed by more than one process at a time [1.2.1].

critical section

a synonym for *critical region*.

data flow diagram (DFD)

a graph that depicts data nodes (data sources, data sinks, data storage, and processes performed on data) and data arcs (data flow connecting data nodes) [2.2.5].

database

a structured collection of data items [3.4].

dataflow machine

a set of processing elements that are triggered to produce a result by the presence of the required operands [1.4.2].

deadlock

a program state in which a process is delayed awaiting an event that will not occur [1.2.2].

degree of granularity

in databases, the size of the data item on which locks are imposed [3.4].

discrete event simulation

a simulation mechanism in which all significant actions have associated events, events are maintained in time order and the model executes by advancing from one event to the next, performing any required computation at each step [3.3].

distributed processing

the sharing of the processors of a multicomputer among a set of competing processes [1.4.2].

distributed program

a parallel program designed to be executed on a network of autonomous processors that do not share main memory [1.1.3].

distributed system

a parallel system executed on a network of autonomous processors that do not share main memory [1.1.4].

embedded system

a hardware/software system with a real-time control program as a component [3.1].

event-driven program

a synonym for *reactive program*.

event flow (Ward/Mellor Real-Time Structured Analysis)

a signal indicating that an event has taken place [2.2.5].

explicitly concurrency

the concurrency present in a program specifying actions that are intended to be executed in parallel [Preface, 1.1.1].

explicit interaction

the interaction among processes of a concurrent program that is specified explicitly [1.1.6].

fairness

(1) (weak) a mechanism for ensuring that when a choice among possible actions is made repeatedly, no action is ignored indefinitely (i.e., there is no indefinite postponement present) [1.2.4, 1.3.2].

(2) (strong) a mechanism for ensuring that when a choice among possible actions is made, each action has an equal probability of selection [1.2.4, 1.4.5].

finite state machine (FSM)

a computational model consisting of a finite number of states and transitions between those states [2.2.4].

fork-and-join concurrency

a model of process creation and termination in which a sequential computation divides into parallel threads of execution that later recombine [1.4.3].

grain of concurrency

the mean computation time between communications during the execution of a concurrent program [1.4.1].

implicitly concurrency

the concurrency present in a program that is designed to be sequential but which includes actions that can be executed in parallel [1.1.1].

implicit interaction

the interaction among processes of a concurrent program that occurs implicitly as a consequence of sharing resources needed for execution [1.1.6].

indefinite postponement

a program state in which a process is delayed awaiting an event that may never occur [1.2.3].

inherent concurrency

the concurrency present in a program constructed to control or model physical systems that involve parallel activity [Preface].

induction

a method of proof in which a property is shown to hold for an initial state of a computation and for each change of state within that computation [1.3.1].

invariant

a condition that is true at all points in a computation [1.3.1].

kernel

(1) that portion of an operating system that is kept in main memory at all times;

(2) a software module that encapsulates an elementary function or functions of a system [1.4.5].

level of concurrency

the mean number of active processes present during the execution of a program [1.4.1].

linear time temporal logic

a version of temporal logic that applies to individual program computations [2.2.1].

liveness

a program property that holds at some point in a computation [1.3.2].

lock

in databases, a mechanism for controlling read and write access to data items; several processes may read a data item simultaneously but only one at a time may modify it [3.4].

lockout

a synonym for *indefinite postponement*.

loose coupling

a network connection between computers [1.4.2].

message passing

a mechanism for enabling one process to make information available to others by directing it to the processes concerned [1.1.6].

monitor

a data structure encapsulating shared variables and defining a set of operations through which the variables may be manipulated; only one process at a time may execute any of the operations [2.2.3.4].

multicomputer

a computer with several processors, each of which has private main memory [1.4.2].

multiprocessing

the sharing of the processors of a multiprocessor among a set of competing processes [1.4.2].

multiprocessor

a computer with several processors sharing a common main memory [1.4.2].

multiprogramming

the sharing of a single processor among a set of competing processes [1.4.2].

multitasking

a synonym for *multiprogramming*.

mutual exclusion

a mechanism for ensuring that only one process at a time performs a specified action [1.2.1].

nondeterminism

a property of a program, in which that there is a partial ordering, rather than a total ordering, on the actions it specifies [1.1.5].

nucleus

a synonym for *kernel*.

optimistic scheduling

a mechanism for controlling concurrent access to a database; free access is permitted and any transaction that causes conflict is rolled back; the database is modified through a two-phase commit procedure [3.4].

parallel program

a concurrent program designed for execution on parallel hardware [1.1.3].

parallel system

a concurrent system executed on parallel hardware [1.1.4].

performance-oriented concurrency

the concurrency present in a program constructed to take advantage of hardware supporting parallel processing [Preface].

Petri net

an abstract formal model of information flow, showing static and dynamic properties of a system. A Petri net is usually represented as a graph having two types of node (called places and transitions) connected by arcs, and markings (called tokens) indicating dynamic properties [2.2.2].

pipeline processor

a set of processing elements dedicated to performing the separate low-level steps of an arithmetic operation in sequence [1.4.2].

process

(1) the execution of a sequential program [1.1.2];

(2) the name of a program construct used to describe the behavior of a process [1.1.2].

pseudoparallel program

a concurrent program designed for execution by a single processor [1.1.3].

quasiparallel program

a pseudoparallel program in which processes execute cooperatively by transferring control to each other using a coroutine mechanism [1.1.3].

reactive program

a program that carries out operations in response to the input it receives [3.1].

ready

a process state in which the process is awake but unable to proceed until it is assigned a processor [1.4.4].

reduction machine

a set of processing elements, each of which is triggered to obtain operands by a request for a result [1.4.2].

rendezvous

the synchronization of two processes to exchange information [2.4.2].

resource

(1) a facility in a computing system;

(2) in the *conditional critical region* mechanism, a construct in which the variables referenced in a *critical region* are identified [2.4.1.3].

result parallelism

a design technique for parallel programs that introduces parallelism in the construction of the data structure produced by a program [2.3].

roll-back

the mechanism for undoing a partial change to a database to restore it to a consistent state [3.4].

running

a process state in which the process is awake and has been assigned a processor [1.4.4].

safety

a program property that holds at every point in a computation [1.3.1].

scale of concurrency

the mean duration of processes in the execution of a program [1.4.1].

scheduler

the software responsible for administering the allocation of processors to processes during the execution of a program [1.4.1].

semaphore

a variable used to control access to a critical region [2.4.1.2].

sequential program

a program specifying statements that are intended to be executed in sequence [1.1.2].

shared resource

a facility of a concurrent program that is accessible to two or more processes of that program [1.1.6.1].

shared variables

a mechanism that enables one process to make information available to others by leaving it in variables accessible to the processes concerned [1.1.6].

specialist parallelism

a design technique for parallel programs that introduces parallelism at the level of autonomous program components [2.3].

spin-lock

the state variables on which *busy waiting* is performed.

starvation

a synonym for *indefinite postponement*.

state formula

a predicate evaluated for a particular state of a computation [2.2.1].

state transition diagram

a graphical representation of a finite state machine in which nodes represent states and arcs represent state transitions [2.2.4].

state transition matrix

a grid representation of a finite state machine in which system states range across and down the grid, and grid elements identify permitted state transitions [2.2.4].

synchronization

(1) the control of the execution of two or more interacting processes so that they perform the same operation simultaneously;

(2) the control of the execution of two or more interacting processes so that they exclude each other from performing the same operation simultaneously [1.1.6].

synchronous communication

message passing between processes achieved by synchronizing their execution to perform the transfer of information from one to another [1.1.6.2].

task

the name of a program construct used to describe the behavior of a process [1.1.2].

temporal logic

an extension of classical logic to deal with time; it is a formalism for specifying structures of states [2.2.1].

temporal operator

a qualification of the range over which an assertion about the state of a computation applies [2.2.1].

terminated

a process state in which the execution of the process is complete [1.4.4].

time-dependent error

see *transient error*.

time slicing

a mode of concurrent program execution in which ready processes of equal priority are allowed to execute in rotation for a small, fixed period of processing time [1.4.5].

trace

a sequence of states and associated events in a computation [2.2.1].

transaction

the set of actions performed on a set of data items, transforming a database from one consistent state to another [3.4].

transient error

an error that may not be repeatable because of nondeterministic program behavior [1.2.6].

two-phase commit

a database transaction mechanism in which the transaction changes are first determined and then committed if they are not in conflict with any other transaction that has completed [3.4].

unit of concurrency

the language component on which program behavior is defined [1.4.1].

vector processor

a pipeline processor that can execute the same instruction on a vector of operands [1.4.2].

workstation-LAN (local area network)

a multicomputer whose processors are in the same vicinity [1.4.2].

workstation-WAN (wide area network)

a multicomputer whose processors are physically remote [1.4.2].

Teaching Considerations

Potential Audience

The material in this curriculum module might be used in a course dealing specifically with concurrent programming or in one where concurrent programming is relevant. The latter case includes courses devoted to specific programming languages and to specific application areas. This defines four main classes of course:

1. A *concurrent programming* course, i.e., a course dedicated to all aspects of concurrent programming.
2. A *related computing* course, i.e., a course requiring a broad introduction to concurrent programming; examples include hardware courses and formal courses on specification and verification.
3. A *language-specific* course, i.e., a course largely devoted to a particular programming language, such as Ada, where facilities for concurrent programming are a major concern.
4 An *application-specific* course, i.e., a course largely devoted to a particular application area, such as operating systems, where the design of concurrent software is a major issue.

Prerequisites

Following the philosophy outlined in the Preface, a concurrent programming course should be the first computing course offered to students! Until this view is more widely shared, it seems likely that those starting a course involving concurrency will already be skilled in the development of sequential programs. Thus, concurrent programming will tend to be taught as a generalization of sequential programming techniques. A report of one experience in this area may be found in [Bustard90].

Using the Outline

This module deals with fundamental material that should be covered in total, regardless of where it is used. The presentation time will vary according to the type of course involved, the depth of coverage required,

the number of examples used, and the experience of the intended audience.

Advice is offered separately for each type of course identified above.

1. Concurrent Programming Course

A concurrent programming course should take a broad view of the subject. In practice, this means covering a range of techniques for each phase of the life cycle and studying several application areas. The chances of achieving generality can be improved by devoting at least a third of the available time to the requirements analysis and design phases of software development.

2. Related Computing Course

Where concurrency is a component of a course, only a few topics can be considered in detail. These will tend to be dictated by the nature of the course. In all cases, however, it seems desirable to give adequate coverage to the software engineering aspects of program development. In practice, this means presenting a suitable software development method in some detail.

3. Language-Specific Course

In a language-specific course, the need to cover details of the language in preparation for practical exercises tends to encourage a rushed treatment of requirements analysis and design. To ensure adequate coverage, it may be necessary to present software development bottom-up; that is, after basic concepts, cover language issues, then design, and finally requirements.

4. Application-Specific Course

In an application-specific course, it is tempting to draw all of the illustrative examples from the application area. This seems undesirable as it tends to deny the generality of the concepts involved. One compromise is to first discuss programming situations by analogy with real-world interactions; this is particularly appropriate when considering the basic concurrency concepts.

Objectives

On covering the material outlined in this module, a student should understand the nature of a concurrent program and the means by which it can be constructed and executed. Such comprehension might be assessed in a written or oral examination.

More specifically, a student should:

- Understand the concepts of sequential process, sequential program and concurrent program.
- Understand the differences between a concurrent program, a parallel program and a distributed program.
- Appreciate the reasons for constructing a program in a concurrent form.
- Be able to relate the process interactions that occur in a concurrent program to real-world resource management and communication situations.
- Understand the problems that are specific to concurrent programs and the means by which such problems can be avoided or overcome.
- Be aware of the different environments in which a concurrent program may be executed and the corresponding means of execution that are appropriate.
- Be aware of the influence of the execution environment on program design.
- Appreciate the role of formality in the description of concurrent programs.
- Understand one or more techniques for specifying concurrent behavior.
- Understand one or more techniques for designing concurrent programs.
- Understand one or more of the implementation languages in which a concurrent program can be expressed.
- Have knowledge of the historical development of mechanisms for controlling process interaction in a concurrent program.
- Appreciate the main characteristics of the common concurrency application areas.

With such comprehension, a student should then be able to:

- Undertake a deeper study of any concurrency-related topic.
- Examine in depth the concurrency aspects of any particular application area.

Resources

There is no single textbook that covers all of the material outlined in this module. Most textbooks have a particular focus, such as an application area, a programming language, the formal expression of program properties, machine architectures, software analysis and design, and so on. References to textbooks with such emphases may be found in the body of the outline, with some further detail presented in the bibliography.

Bibliography

The table of contents is shown for each book that is considered particularly relevant to the material covered in this module, with the number of pages in a chapter given in brackets after each chapter heading.

Andrews83

Andrews, G. R., and Schneider, F. B. "Concepts and Notations for Concurrent Programming." *ACM Computing Surveys 15*, 1 (Mar. 1983), 3-43. Reprinted in [Gehani88].

Abstract: *Much has been learned in the last decade about concurrent programming. This paper identifies the major concepts of concurrent programming and describes some of the more important language notations for writing concurrent programs. The roles of processes, communication and synchronization are discussed. Language notations for expressing concurrent execution and for specifying process interaction are surveyed. Synchronization primitives based on shared variables and on message passing are described. Finally, three general classes of concurrent programming languages are identified and compared.*

Major developments since the appearance of this paper are surveyed in [Bal89].

Bal89

Bal, H. E., Steiner, J. G., and Tanenbaum, A. S. "Programming Languages for Distributed Computing Systems." *ACM Computing Surveys 21*, 3 (Sept. 1989), 261-322.

Abstract: *When distributed systems first appeared, they were programmed in traditional sequential languages, usually with the addition of a few library procedures for sending and receiving messages. As distributed applications became more commonplace and more sophisticated, this ad hoc approach became less satisfactory. Researchers all over the world began designing new programming languages specifically for implementing distributed applications. These languages and their history, their underlying principles, their design, and their use are the subject of this paper.*

We begin by giving our view of what a distributed system is, illustrating with examples to avoid confusion on this important and controversial point. We then describe the three main characteristics that distinguish distributed programming languages from traditional sequential languages, namely, how

they deal with parallelism, communication, and partial failures. Finally, we discuss 15 representative distributed languages to give the flavor of each. These examples include languages based on message passing, rendezvous, remote procedure call, objects, and atomic transitions, as well as functional languages, logic languages, and distributed data structure languages. The paper concludes with a comprehensive bibliography listing over 200 papers on nearly 100 distributed programming languages.

This is a very useful reference document. Note that the paper deals with concurrency in general, despite its title. The only area not covered is shared memory systems, details of which may be found in [Andrews83].

Bamberger89

Bamberger, J., Colket, C., Firth, R., Klein, D., and Van Scoy, V. *Kernel Facilities Definition (Version 3.0)*. Technical Report CMU/SEI-88-TR-16, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1989.

Table of Contents:
1. Kernel Background: rationale, definitions, kernel functional areas; 2. Requirements: general, processor, process, semaphore, scheduling, communication, interrupt, time, alarm, tool; 3. Kernel Primitives (matching requirements).

This report defines the functionality of a distributed Ada real-time kernel.

Ben-Ari82

Ben-Ari, M. *Principles of Concurrent Programming*. Prentice-Hall International, 1982.

Table of Contents:
1. What is Concurrent Programming? (17); 2. The Concurrent Programming Abstraction (11); 3. The Mutual Exclusion Problem (21); 4. Semaphores (23); 5. Monitors (20); 6. The Ada Rendezvous (16); 7. The Dining Philosophers (10).

A short but highly informative introduction to the basic concepts in concurrent programming. All the major classical examples from the literature are discussed. One distinctive feature of the book is that it introduces a rigorous approach to the analysis of concurrent behavior without resorting to mathematical notation.

An appendix provides an implementation kit for the simple concurrent programming language Co-Pascal. The exercises are good.

Ben-Ari90

Ben-Ari, M. *Principles of Concurrent & Distributed Programming*. Prentice-Hall International, 1990.

Just published. Details not yet available.

Bernstein87

Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*. Reading, M.A.: Addison-Wesley, 1987.

Berztiss87

Berztiss, A. *Formal Specification of Software.* Curriculum Module SEI-CM-8-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Oct. 1987.

Abstract: *This module introduces methods for the formal specification of programs and large software systems, and reviews the domains of application of these methods. Its emphasis is on the functional properties of software. It does not deal with the specification of programming languages, the specification of user-computer interfaces, or the verification of programs. Neither does it attempt to cover the specification of distributed systems.*

Birtwistle79

Birtwistle, G. M. *Discrete Event Modeling on SIMULA.* London: Macmillan, 1979.

This is an introductory level text containing many examples in Simula (credited as the first object-oriented language).

Booch86

Booch, G. "Object-Oriented Development." *IEEE Trans. Software Eng. SE-12*, 2 (Feb. 1986), 211-221.

Abstract: *Object-oriented development is a partial-lifecycle software development method in which the decomposition of a system is based upon the concept of an object. This method is fundamentally different from traditional functional approaches to design and serves to help manage the complexity of massive software-intensive systems. The paper examines the process of object-oriented development as well as the influences upon this approach from advances in abstraction mechanisms, programming languages and hardware. The concept of an object is central to object-oriented development and so the properties of an object are discussed in detail. The paper concludes with an examination of the mapping of object-oriented techniques to Ada using a design case study.*

Brackett90

Brackett, J. W. *Software Requirements.* Curriculum Module SEI-CM-19-1.2, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Jan. 1990.

Abstract: *This curriculum module is concerned with the definition of software requirements – the software engineering process of determining what is to be produced – and the products generated in that definition. The process involves all of the following: requirements identification, requirements analysis, requirements representation, requirements*

communication and development of acceptance criteria and procedures. The outcome of requirements definition is a precursor of software design.

Brinch Hansen77

Brinch Hansen, P. *The Architecture of Concurrent Programs.* Englewood Cliffs, N. J.: Prentice-Hall, 1977.

This is an edited collection of papers that describe the monitor-based programming language Concurrent Pascal and its use in a number of applications; the most significant example is the Solo Operating System.

Brinksma88

Brinksma, E., ed. *Information Processing Systems - OSI - LOTOS - A Formal Technique Based on Temporal Ordering of Observational Behavior.* Standard ISO IS 8807, 1988.

This standard for the formal description language LOTOS includes a tutorial introduction to the notation.

BSI89

BSI. *Draft British Standard for Programming Language Modula 2: Third Working Draft.* Standard ISO/IEC DP 10514, British Standards Institute, Nov. 1989.

Defines a "Processes" and a "Semaphores" module. The semaphore model makes no allowance for the waiting time of processes that are delayed.

Burns85

Burns, A. *Concurrent Programming in Ada.* Cambridge: Cambridge University Press, 1985.

Table of Contents:
1. The Ada Language (16); 2. The Nature and Uses of Concurrent Programming (10); 3. Inter-Process Communication (22); 4. Ada Task Types and Objects (12); 5. Ada Inter-Task Communication (16); 6. The Select Statement (22); 7. Task Termination, Exceptions and Attributes (12); 8. Tasks and Packages (14); 9. Access Types for Tasks (10); 10. Resource Management (20); 11. Task Scheduling (12); 12. Low-Level Programming (16); 13. Implementation of Ada Tasking (16); 14. Portability (4); 15. Programming Style for Ada Tasking (12); 16. Formal Specifications (8); 17. Conclusion (6).

Although this book focuses on one particular language supporting concurrent programming, it provides an introduction to other notations and to wider issues in program construction, such as formal specification. No exercises are included.

Bustard88

Bustard, D. W., Elder, J. W. G., and Welsh, J. *Concurrent Program Structures.* Prentice-Hall International, 1988.

Table of Contents:
1. Introduction to Concurrency (12); 2. Execution of Concurrent Programs (11); 3. Design of Concurrent Programs (20); 4. Representation of Concurrent Programs (30); 5. Testing Concurrent Programs (21); 6. Resource Management (46); 7. Communication Management (15); 8. Discrete Event Simulation (32); 9. Real-Time Systems (23); 10. General Purpose Operating Systems (24); 11. Representation of Process Interaction: other approaches (25).

This book provides an introduction to the design and implementation of concurrent programs. Particular emphasis is given to techniques for managing process interaction in different circumstances, illustrated using the shared variable communication model. An appendix gives solutions to the exercises set.

Bustard90

Bustard, D. W. "An Experience of Teaching Concurrency: Looking Forward, Looking Back." *CSEE '90 Fourth SEI Conference on Software Engineering Education*, Lionel Deimel, ed. Springer-Verlag, Apr. 1990.

Abstract: *The book Concurrent Program Structures [Bustard88] was based on a course, Concurrent Systems, introduced at Queen's University, Belfast in 1981. The purpose of this paper is to examine the successful and less successful aspects of that course, with a view to making improvements to the material presented.*

Carriero89

Carriero, N., and Gelernter, D. "How to Write Parallel Programs: A Guide to the Perplexed." *ACM Computing Surveys 21*, 3 (Sept. 1989), 323-357.

Abstract: *We present a framework for parallel programming, based on three conceptual classes for understanding parallelism and three programming paradigms for implementing parallel programs. The conceptual classes are result parallelism, which centers on parallel computation of all elements in a data structure; agenda parallelism, which specifies an agenda of tasks for parallel execution; and specialist parallelism, in which specialist agents solve problems cooperatively. The programming paradigms center on live data structures that transform themselves into result data structures; distributed data structures that are accessible to many processes simultaneously; and message passing, in which all data objects are encapsulated within explicitly communicating processes.*

Chandy88

Chandy, K. M., and Misra, J. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

This is a very thorough, though specific treatment of the theory of concurrent programming.

The book has the following stated goal: "The thesis of this book is that the unity of the programming task transcends differences between the

architectures on which programs can be executed and the application domains from which problems are drawn. Our goal is to show how programs can be developed systematically for a variety of architectures and applications. The foundation, on which program development is based, is a simple theory: a model of computation and an associated proof system."

Coffman71

Coffman, E. G., Elphick, M. J., and Shoshani, A. "System Deadlocks." *ACM Computing Surveys 3*, 2 (June 1971), 67-78.

Abstract: *A problem of increasing importance in the design of large multiprogramming systems is the, so-called, deadlock or deadly-embrace problem. In this article we survey the work that has been done on the treatment of deadlocks from both the theoretical and practical points of view.*

The main significance of this paper is that it defines four necessary conditions for deadlock to occur.

Collofello88

Collofello, J. *Introduction to Software Verification and Validation*. Curriculum Module SEI-CM-13-1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1988.

Abstract: *Software verification and validation techniques are introduced and their applicability discussed. Approaches to integrating these techniques into comprehensive verification and validation plans are also addressed. This curriculum module provides an overview needed to understand in-depth curriculum modules in the verification and validation area.*

Crichlow88

Crichlow, J. M. *An Introduction to Distributed and Parallel Computing*. Prentice-Hall International, 1988.

Table of Contents:
1. Introduction (14); 2. Computer Organization for Parallel and Distributed Computing (31); 3. Communications and Computer Networks (36); 4. Operating Systems for Distributed and Parallel Computing (30); 5. Servers in the Client-Server Network Model (24); 6. Distributed Database Systems (30); 7. Parallel Programming Languages (25).

This book provides a broad bottom-up introduction to parallel processing, starting with a review of the types of hardware available, moving up through communication protocols to operating system and database applications.

Date86

Date, C. J. *An Introduction to Database Systems*. Addison-Wesley, 1986. (2 volumes).

This is a general introduction to database design and management. One chapter in Volume II is devoted to concurrency.

Davis88

Davis, A. M. "A Comparison of Techniques for the Specification of External System Behavior." *Comm. ACM 31*, 9 (Sept. 1988), 1098-1115.

Davis summarizes and compares 11 techniques for specifying behavior, including finite state machines, statecharts and Petri nets.

Deitel84

Deitel, H. M. *An Introduction to Operating Systems*. Reading, Mass.: Addison-Wesley, 1984.

Table of Contents:
There are 22 chapters divided into 8 parts. The chapters most relevant to concurrency are: 2. Process Concepts (20); 3. Asynchronous Concurrent Processes (21); 4. Concurrent Programming: monitors, the Ada rendezvous (24); 6. Deadlock (28); 10. Job and Processor Scheduling (22); 11. Multiprocessing (32); 16. Network Operating Systems (30).

This a very popular text on operating systems. It is comprehensive and well organized, and the material is clearly presented.

Dijkstra68

Dijkstra, E. W. "Cooperating Sequential Processes." *Programming Languages*, F. Genuys, ed. Academic Press, 1968, 43-112.

This classic paper in concurrent programming is divided into six sections: 1. On the Nature of Sequential Processes; 2. Loosely Connected Processes; 3. The Mutual Exclusion Problem Revisited; 4. The General Semaphore; 5. Cooperation via Status Variables; 6. The Problem of Deadly Embrace.

Dijkstra71

Dijkstra, E. W. "Hierarchical Ordering of Sequential Processes." *Acta Informatica 1* (1971), 115-138.

This paper deals largely with operating system design but also examines the general issue of mutual exclusion, with examples. It is also where the classic problem of the Dining Philosophers was first presented.

Feldman90

Feldman, M. *Language and System Support for Concurrent Programming*. Curriculum Module SEI-CM-25, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Apr. 1990.

Abstract: *This curriculum module is concerned with support for concurrent programming provided to the application programmer by operating systems and programming languages. This includes system calls and language constructs for process creation, termination, synchronization, and communication. Several readily-available languages are discussed and compared; concurrent programming using system services of the UNIX operating system is introduced for the sake of comparison and contrast.*

Ford86

Ford, G. A., and Weiner, R. S. *Modula-2: A Software Development Approach*. New York: Wiley, 1986.

Modula-2 provides a small set of primitives on which a process abstraction module can be built. This book describes these primitives and from them develops a series of standard modules for use in concurrent programming. (The majority of the book is devoted to the development of sequential programs). See also [BSI89].

Galton87

Temporal Logics and Their Applications. Anthony Galton, ed. Academic Press, 1987.

This book was derived from a conference on Temporal Logic and Its Applications, held in 1986. The first chapter gives a broad introduction to the use of temporal logic in computer science and the second deals with the use of temporal logic in the specification of concurrent systems.

Gehani84

Gehani, N. *Ada: Concurrent Programming*. Englewood Cliffs: Prentice-Hall, 1984.

Table of Contents:
1. Concurrent Programming: a quick survey (28); 2. Tasking Facilities (52); 3. Task Types (18); 4. Exceptions and Tasking (10); 5. Device Drivers (12); 6. Real-Time Programming (20); 7. Some Issues in Concurrent Programming (20); 8. More Examples (20); 9. Some Concluding Remarks (6).

Gehani is mainly concerned with explaining the concurrency features of Ada but also tackles some general issues. A large number of examples are used but very few exercises are provided.

Gehani88

Concurrent Programming. N. Gehani and A. D. McGettrick, eds. Addison-Wesley, 1988.

Table of Contents:
Organized into five sections, each of which is introduced briefly. There is no index. The summary that follows shows the number of papers in each section and the total page length involved.
1. Survey of Concurrent Programming (1:70); 2. Concurrent Programming Languages (4:90); 3. Concurrent Programming Models (8:188); 4. Assessment of Concurrent Programming Languages (9:216); 5. Concurrent Programming Issues (2:21).

This book brings together a collection of papers that deal mostly with language issues in concurrent programming.

Gehani89

Gehani, N., and Roome, W. D. *Concurrent C.* Summit, N. J.: Silicon Press, 1989.

Table of Contents:
1. Basics (32); 2. Advanced Facilities (46); 3. Run-time Environment (14); 4. Large Examples (40); 5. Concurrent C++ (20); 6. Concurrent Programming Models (14); 7. Concurrent Programming Issues (22); 8. Discrete Event Simulation (22); Appendix: Comparison with Ada (and other topics).

This book is a programmer's guide to a set of concurrency extensions to the programming language C. Alternative programming models and basic concurrency concepts are also considered.

Gelernter88

Gelernter, D. "Parallel Programming: Experiences with Applications, Languages and Systems." *Sigplan Notices 23*, 9 (Sept. 1988). ACM/Sigplan PPEALS.

This is the proceedings of an annual conference on parallel programming. The foreword likens worries about the difficulty of parallel programming to the initial reaction against Talking Pictures!

Gomma87

Gomma, H. "Using the DARTS Software Design Method for Real-Time Systems." *Proc. 12th Structured Methods Conf.* Chicago: Structured Techniques Association, Aug. 1987, 76-90.

Abstract: *The paper describes a software design method for real-time systems and gives an example of its use. The method is called DARTS, the Design Approach for Real-Time Systems.*

This paper describes how DARTS can be used in conjunction with Real-Time Structured Analysis [Ward85].

Gomma89

Gomma, H. *Software Design Methods for Real-Time Systems*. Curriculum Module SEI-CM-22-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1989.

Abstract: *This module describes the concepts and methods used in the software design of real-time systems. It outlines the characteristics of real-time systems, describes the role of software design in real-time system development, surveys and compares some software design methods for real-time systems, and outlines techniques for the verification and validation of real-time designs. For each design method treated, its emphasis, concepts on which it is based, steps used in its application, and an assessment of the methods are provided.*

Habermann76

Habermann, A. N. *Introduction to Operating System Design*. Science Research Associates, Inc., 1976.

This is a thorough, classic treatment of operating systems, concentrating on principles of design rather that details of specific operating systems or implementation languages.

Harel88

Harel, D. "On Visual Formalisms." *Comm. ACM 31*, 5 (May 1988), 514-530.

Harel introduces the general concept of a "higraph" and shows its application in the extension of state-transition diagrams to statecharts that can describe concurrent systems.

Hoare74

Hoare, C. A. R. "Monitors: An Operating System Structuring Concept." *Comm. ACM 17*, 10 (1974), 549-557. Reprinted in [Gehani88].

This paper introduces the monitor concept and demonstrates its use in a series of examples.

Hoare78

Hoare, C. A. R. "Communicating Sequential Processes." *Comm. ACM 21*, 8 (Aug. 1978), 666-677. Reprinted in [Gehani88].

An earlier (and simpler) version of the notation described in [Hoare85].

Hoare85

Hoare, C. A. R. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

Table of Contents:
1. Processes (42); 2. Concurrency (36); 3. Nondeterminism (32); 4. Communication (38); 5. Sequential Processes (26); 6. Shared Resources (26); 7. Discussion (28).

This book gives a mathematical treatment to the description of concurrent behavior. The formalism is developed gradually through a series of small examples and linked to programming language concepts in the final chapter. Note that the notation used differs in some respects from that described in [Hoare78].

Holt83

Holt, R. C. *Concurrent Euclid, The UNIX System and TUNIS*. Reading, Mass.: Addison-Wesley, 1983.

Table of Contents:
1. Concurrent Programming and Operating Systems (16); 2. Concurrency

Problems and Language Features (42); 3. Concurrent Euclid: sequential features (34); 4. Concurrent Euclid: concurrency features (22); 5. Examples of Concurrent Programs (30); 6. UNIX: User Interface and File System (18); 7. UNIX: User Processes and the Shell (14); 8. Implementation of the UNIX Nucleus (18); 9. TUNIS: A UNIX Compatible Nucleus (18); 10. Implementing a Kernel (30).

Largely a textbook dealing with the concurrency aspects of operating system design, this book also introduces the monitor-based programming language Concurrent Euclid.

Hughes88

Hughes, J. G. *Database Technology: A Software Engineering Approach*. Prentice-Hall International, 1988.

This is a book on the design of relational database systems, with implementation details illustrated in Modula-2. It contains a chapter on the management of concurrent access to a database.

Hwang84

Hwang, K., and Briggs, F. A. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1984.

Table of Contents:
1. Introduction to Parallel Processing (51); 2. Memory and Input-Output Subsystems (93); 3. Principles of Pipelining and Vector Processing (88); 4. Pipeline Computers and Vectorization Methods (92); 5. Structures and Algorithms for Array Processors (68); 6. SIMD Computers and Performance Enhancement (66); 7. Multiprocessor Architecture and Programming (98); 8. Multiprocessor Control and Algorithms (86); 9. Example Multiprocessor Systems (89); 10. Data Flow Computers and VLSI Computations (81).

A comprehensive coverage of the field at the time of publication. The bibliography is extensive.

IEEE83

IEEE, *IEEE Standard Glossary of Software Engineering Terminology*. ANSI/IEEE Std 729-1983, 1983.

INMOS88

occam 2 Reference Manual. Prentice-Hall International, 1988.

This is a definition and tutorial guide to the programming language occam. The design of this language is strongly influenced by Hoare's CSP notation [Hoare85].

Joseph84

Joseph, M., Prasad, V. R., and Natarajan, N. *A Multiprocessor Operating System*. Prentice-Hall International, 1984.

This book contains a detailed presentation of the structure and code of an operating system for a multiprocessor. The implementation language, CCN Pascal, is monitor-based [Hoare74].

Kreutzer86

Kreutzer, W. *System Simulation Programming Styles and Languages.* Addison-Wesley, 1986.

This is a simulation text designed for computer scientists. It covers a range of techniques and is well illustrated. The bibliography is good. (An interesting cartoon introduces each chapter!)

Kuhn81

Tutorial on Parallel Processing. R. H. Kuhn and D. A. Padua, eds. IEEE Computer Society Press, 1981.

This is a collection of some of the most significant papers on parallel processing to appear in the 1970s. About half of the papers are on hardware topics and half deal with software (46 in total).

Lamport83

Lamport, L. "Specifying Concurrent Program Modules." *ACM Trans. Prog. Lang. Syst. 5*, 2 (Apr. 1983), 190-222.

Abstract: *A method for specifying program modules in a concurrent program is described. It is based upon temporal logic, but uses new kinds of temporal assertions to make the specifications simpler and easier to understand. The semantics of the specification is described informally, and a sequence of examples are given culminating in a specification of three modules comprising the alternating-bit communication protocol. A formal semantics is given in the appendix.*

Lamport89

Lamport, L. "A Simple Approach to Specifying Concurrent Systems." *Comm. ACM 32*, 1 (Jan. 1989), 32-45.

Abstract: *Over the past few years, I have developed an approach to the formal specification of concurrent systems that I now call the transition axiom method. The basic formalism is described in [Lamport83], but the formal details tend to obscure the important concepts. Here, I attempt to explain these concepts without discussing the details of the underlying formalism.*

This paper is organized as a series of questions and answers, addressing such fundamental questions as "what is a formal specification?" and "what is a system?" It gives particular emphasis to the concepts of safety and liveness properties.

Lister84

Lister, A. M. *Fundamentals of Operating Systems (3rd Edition).* Springer-Verlag, 1984.

This is a long-standing text, demonstrating that the concepts involved are changing little with the technology advances. One distinctive feature of the book is that it discusses the structure of an operating system in layers, from the nucleus (kernel) up to job control.

McDowell89

McDowell, C. E., and Helmbold, D. P. "Debugging Concurrent Programs." *ACM Computing Surveys 21*, 4 (Dec. 1989), 593-622.

Abstract: *The main problems associated with debugging concurrent programs are increased complexity, the "probe effect", nonrepeatability, and the lack of a synchronized global clock. The probe effect refers to the fact that any attempt to observe the behavior of a distributed system may change the behavior of that system. For some parallel programs, different executions with the same data will result in different results even without any attempt to observe the behavior. Even when the behavior can be observed, in many systems the lack of a synchronized clock makes the results of the observation difficult to interpret. This paper discusses these and other problems related to debugging concurrent programs and presents a survey of current techniques. Systems using three general techniques are described: traditional or breakpoint style debuggers, event monitoring systems and static analysis systems. In addition, techniques for limiting, organizing and displaying a large amount of data produced by the debugging systems are discussed.*

Milner89

Milner, R. *Communication and Concurrency*. Prentice-Hall International, 1989.

This book describes a mathematical theory of communicating systems, referred to here as a "process calculus." A large part of the book is concerned with equivalence of formal descriptions – the basis of verification.

Morell89

Morell, L. J. *Unit Testing and Analysis*. Curriculum Module SEI-CM-9-1.2, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Apr. 1989.

Abstract: *This module examines the techniques, assessment, and management of unit testing and analysis. Testing and analysis strategies are categorized according to whether their coverage goal is functional, structural, error-oriented, or a combination of these.*

Parnas84

Parnas, D., Clements, P., and Weiss, D. "The Modular Structure of Complex Systems." *Proc. 7th Intl. Conf. Software. Eng., Long Beach, California*. IEEE Computer Society, 1984, 408-416.

Abstract: *This paper discusses the organization of software that is inherently complex because there are very many arbitrary details that must be precisely right for the software to be correct. We show how the software design technique known as information hiding or abstraction can be*

supplemented by a hierarchically-structured document, which we call a module guide. The guide is intended to allow both designers and maintainers to identify easily the parts of the software that they must understand without reading irrelevant details about other parts of the software. The paper includes an extract from a software module guide to illustrate our proposals.

Perrott88

Perrott, R. H. *Parallel Programming.* Addison-Wesley, 1988.

Table of Contents:
HISTORY AND DEVELOPMENT 1. Hardware Technology Developments (13); 2. Software Technology Developments (9). ASYCHRONOUS PARALLEL PROGRAMMING 3. Mutual Exclusion (12); 4. Process Synchronization (11); 5. Message Passing Primitives (13); 6. Modula-2 (13); 7. Pascal Plus (15); 8. Ada (14); 9. occam: a distributed computing language (16). SYNCHRONOUS PARALLEL PROGRAMMING Detection of Parallelism Languages: 10. Cray-1 FORTRAN Translator (20); 11. CDC Cyber Fortran (13); Expression of Machine Parallelism Languages: 12. Illiac4 CDF FORTRAN (13); 13. Distributed Array Processor FORTRAN (13). Expression of Problem Parallelism Languages: 14. ACTUS: a Pascal based language (24). DATA FLOW PROGRAMMING 15. Data Flow Programming (21).

This broad introduction to concurrency covers a wide range of programming language representations and machine architectures.

Peterson81

Peterson, J. *Petri Net Theory and the Modeling of Systems.* Englewood Cliffs, N. J.: Prentice-Hall, 1981.

This is a readable introduction to Petri nets and their application. See also [Reisig85].

Pneuli86

Pneuli, A. "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Recent Trends." *Current Trends in Concurrency,* J. W. de Bakker et al, ed. New York: Springer-Verlag, 1986, 510-584.

This long paper summarizes the work by Pneuli and others relating to the specification and verification of reactive systems using temporal logic. The paper is organized in four parts. Temporal logic is introduced in the second part, the first dealing with various abstract and concrete computational models.

Polychronopoulos88

Polychronopoulos, C. D. *Parallel Programming and Compilers.* Boston: Kluwer Academic Publishers, 1988.

Table of Contents:
1. Parallel Architectures and Compilers (14); 2. Program Restructuring for Parallel Execution (67); 3. A Comprehensive Environment for Automatic

Packaging and Scheduling of Parallelism (31); 4. Static and Dynamic Loop Scheduling (50); 5. Run-Time Overhead (15); 6. Static Program Partitioning (17); 7. Static Task Scheduling (21); 8. Speedup Bounds for Parallel Programs (13).

This is an advanced text on parallel processing that assumes a basic knowledge of the subject area.

Raynal86

Raynal, M. *Algorithms for Mutual Exclusion*. Cambridge, Mass.: The MIT Press, 1986.

Table of Contents:
1. The Nature of Control Problems in Parallel Processing (16); 2. The Mutual Exclusion Problem in a Centralized Framework: Software Solutions (22); 3. The Mutual Exclusion Problem in a Centralized Framework: Hardware Solutions (10); 4. The Mutual Exclusion Problem in a Distributed Framework: Solutions Based on State Variables (16); 5. The Mutual Exclusion Problem in a Distributed Framework: Solutions Based on Message Communication (22); 6. Two Further Control Problems (12).

This is a comprehensive collection of the algorithms associated with mutual exclusion in concurrent systems covering both shared memory and distributed systems.

This is a translation of a book first published in French in 1984.

Reisig85

Reisig, W. *Petri Nets: An Introduction*. Springer-Verlag, 1985.

This is a short systematic introduction to Petri nets. It has an extensive bibliography. See also [Peterson81].

Rombach90

Rombach, H. D. *Software Specifications: A Framework*. Curriculum Module SEI-CM-11-2.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Jan. 1990.

Abstract: *This curriculum module presents a framework for understanding software product and process specifications. An unusual approach has been chosen in order to address all aspects related to "specification" without confusing the many existing uses of the term. In this module, the term specification refers to any plan (or standard) according to which products of some type are constructed or processes of some type are performed, not to the products or processes themselves. In this sense, a specification is itself a product that describes how products of some type should look or how processes of some type should be performed. The framework includes: a reference software life-cycle model and terminology, a characterization scheme for software product and process specifications, guidelines for using the characterization scheme to identify clearly certain life cycle phases, and guidelines for using the characterization scheme to select and evaluate specification techniques.*

Scacchi87

Scacchi, W. *Models of Software Evolution: Life Cycle and Process*. Curriculum Module SEI-CM-10-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Oct. 1987.

Abstract: *This module presents an introduction to models of software evolution and their role in structuring software development. It includes a review of traditional life cycle models as well as software process models that have been recently proposed. It identifies three kinds of alternative models of software evolution that focus attention on either the products, production processes, or production settings as the major source of influence. It examines how different software engineering tools and techniques can support life cycle or process approaches. It also identifies techniques for evaluating the practical utility of a given model of software evolution for development projects in different kinds of organizational settings.*

The material presented in this module is equally applicable to the development of sequential and concurrent software.

Schiper89

Schiper, A. *Concurrent Programming*. Halsted Press; John Wiley & Sons Inc., 1989.

Table of Contents:
1. Introduction (7); 2. Input/Output and Interrupts (9); 3. The Process Concept (10); 4. Mutual Exclusion (18); 5. Cooperation Between Processes (18); 6. Portal and Monitors (20); 7. Modula-2 and Kernels (23); 8. Ada and Rendezvous (22); 9. An Example of Designing a Concurrent Program (24).

This is a translation from a 1986 text *Programmation Concurrente*. It provides a general introduction to concurrency topics but tends to give emphasis to the execution of concurrent programs by a single processor.

Sutcliffe88

Sutcliffe, A. *Jackson System Development*. Prentice-Hall International, 1988.

The early stages of the Jackson System Design technique develop a system as a collection of interacting processes. The technique is used most commonly for sequential (data processing) applications, so the concurrent representation is usually converted into a sequential form by a procedure known as "process inversion."

Swartout86

Swartout, W., and Balzer, R. “On the Inevitable Intertwining of Specification and Implementation.” *Software Specification Techniques*, Narhain Gehani; Andrew D. McGettrick, eds. Addison-Wesley, 1986, 41-45.

This is a fascinating short paper which suggests that, typically, some implementation issues are decided before specification is complete and that this situation is inevitable.

Tanenbaum85

Tanenbaum, A. S., and van Renesse, R. "Distributed Operating Systems." *ACM Computing Surveys 17*, 4 (Dec. 1985), 419-470.

Abstract: *This paper is intended as an introduction to distributed operating systems, and especially to current university research about them. After a discussion of what constitutes a distributed operating system and how it is distinguished from a computer network, various key design issues are discussed. Then several examples of current research projects are examined in some detail, namely, the Cambridge Distributed Computing System, Amoeba, V, and Eden.*

Treleaven82

Treleaven, P. C., Brownbridge, D. R., and Hopkins, R. P. "Data-Driven and Demand-Driven Computer Architectures." *ACM Computing Surveys 14*, 1 (Mar. 1982), 93-143.

Abstract: *Novel data-driven and demand-driven computer architectures are under development in a large number of laboratories in the United States, Japan, and Europe. These computers are not based on the traditional von Neumann organization; instead, they are attempts to identify the next generation of computer. Basically, in data-driven (e.g. data-flow) computers the availability of operands triggers the execution of the operation to be performed on them, whereas in demand-driven (e.g. reduction) computers the requirement for a result triggers the operation that will generate it. The aim of this paper is to identify the concepts and relationships that exist both within and between the two areas of research.*

Ullman82

Ullman, J. D. *Principles of Database Systems, 2nd Edition*. London: Pitman, 1982.

This is a thorough treatment of database organization and access. The final chapter deals with concurrent database operations.

Ward85

Ward, P. T., and Mellor, S. J. *Structured Development for Real-Time Systems*. Yourdon Press, 1985. (three volumes).

Table of Contents:
Three volumes: 1. Introduction & Tools; 2. Essential Modelling Techniques; 3. Implementation Modelling Techniques.

This book describes real-time program development method in detail, with illustrations.

Wegner89

Wegner, P. "Programming Language Paradigms." *A C M Computing Surveys 21,* 3 (Sept. 1989), 251-510.

Abstract: *There are four papers: 1. Programming Languages for Distributed Computing Systems (52); 2. How to Write Parallel Programs: a Guide to the Perplexed (36); 3. Conception, Evolution and Application of Functional Programming Languages (53); 4. The Family of Concurrent Logic Programming Languages (98).*

Concurrency emerged as the common theme across all four papers in this special issue. Comments on the first two papers are given separately in [Bal89; Carriero89].

Whiddett87

Whiddett, D. *Concurrent Programming for Software Engineers.* Chichester: Ellis Horwood, 1987.

Table of Contents:
THE BASICS 1. The Concept of a Process (31); 2. Process Coordination (35). THE MODELS 3. Procedure Based Interaction: monitors (24); 4. Message Based Interaction (26); 5. Operation Oriented Programs (25); 6. Comparison of Methods (11). THE PRAGMATICS 7. Interfacing to Peripheral Devices (26); 8. Programming Distributed Systems (24).

This book is intended to provide an introduction to concurrent programming for engineers and engineering students; it places greatest emphasis on notation.

Wood89

Wood, D. P., and Wood, W. G. *Comparative Evaluations of Four Specification Methods for Real-Time Systems.* Technical Report CMU/SEI-89-TR-36, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1989.

This report compares four of the most commonly used real-time specification methods. All belong to the family of Real-Time Structured Analysis techniques [e.g., Ward85].

Yourdon89

Yourdon, E. *Modern Structured Analysis.* Prentice-Hall, 1989.

This is a large modern text on Structured Analysis that includes coverage of the real-time extensions to this analysis and design method.