Distributed Software Engineering

Ugo Buy and Sol Shatz
University of Illinois
Chicago, Illinois

1 INTRODUCTION AND BACKGROUND

Distributed software engineering is an area of software engineering that is concerned with the development and the maintenance of reliable software for distributed systems. These systems are characterized by the presence of units that can be executed in parallel with respect to each other. In addition, these units communicate with one another by passing messages over a communication network rather than by sharing structures stored in common areas of memory.

This characterization of a distributed system is deliberately general and reflects the fact that this term is often used with different meanings. This article specifically distinguishes between physically distributed and logically distributed software systems. In a physically distributed system different portions of the system are actually executed on different processors that do not share primary memory. Examples of hardware architectures for physically distributed systems include wide area networks (WANs), local area networks (LANs), and certain kinds of parallel architectures whose processing elements do not share memory, such as multicomputers [Ath88].

In a logically distributed system the software is specified as a collection of parallel units that communicate by passing messages rather than by accessing and modifying a common set of data structures. Note that a logically distributed system need not be physically distributed, in that the parallel units contained in the system could be executed on the same time-shared processor. The opposite is also true: a program that is logically nondistributed, such as a sequential program, might be executed on a multicomputer with the aid of a parallelizing compiler.

In this article, the discussion specifically addresses the issues underlying the development of physically distributed systems, regardless of whether such systems are also logically distributed. In particular, the impact of distribution on the various phases of the software life cycle is analyzed.

Various kinds of parallel units have been used for distributed software. Examples of such units are processes [Bri78], tasks [Uni83], resources [And88], and concurrent objects [Bla87]. For sake

of convenience, in this article the term "process" is used to denote a parallel unit, although the discussion is generally applicable to the other kinds of units as well.

The development of distributed software can be motivated by several goals. For example, some developers may use distributed software to improve system performance by exploiting a distributed architecture. By performing multiple computations simultaneously, distributed systems can take advantage of potential parallelism inherent in a given application.

An alternative reason for developing distributed software is the development of fault-tolerant systems. These systems often perform critical tasks, such as aircraft navigation or patient monitoring, that should not be disrupted by hardware or software failures. To achieve fault-tolerance, system computations and data structures are sometimes replicated across a distributed system. When a failure occurs, computations disrupted by the failure are carried out by surviving processors. Voting schemes are often used to resolve conflicts among units performing a given computation. A discussion of techniques for achieving fault-tolerance is beyond the scope of this article, however.

A final reason for developing distributed software is that the nature of a given application or processing infrastructure may explicitly lead to a distributed implementation. Many Internet and web-based applications have become a driving force for this type of distributed software development. Example domains for such systems include mobile computing and e-commerce. Often, such systems involve geographically distributed databases and allow for computations to be performed at different sites depending on the availability of data and computing resources at each site.

The design, implementation, and testing of distributed software systems are significantly more difficult than in the case of traditional software. There are several reasons for this. First of all, in distributed systems the presence of multiple, asynchronous processes introduces the possibility of such undesirable situations as deadlock and starvation. In a deadlock state, there exists a set of processes that are blocked while mutually waiting for each other to perform some action. A process starves when it is indefinitely prevented from making progress by the lack of a required resource, although the resource may never become permanently unavailable.

An additional difficulty is that the behavior of a distributed system often depends on such arbitrary factors as random delays over communication channels and the unpredictable occurrence of interrupts. These factors and the resulting nondeterminism make design and testing of distributed software extremely difficult.

To complicate matters even further, the number of nondeterministic interleavings of computation sequences performed by asynchronous processes is usually very large. In general, traditional testing and simulation techniques cannot capture all such interleavings in a reasonable amount of

time. Finally, some specific design issues, such as the allocation of processors and of communication channels to logical processes, must often be addressed by designers of distributed systems.

This article is organized as follows. In the next section, approaches for modeling and for specifying distributed software are surveyed. This is followed by a discussion of design issues and an overview of languages for distributed software. Then, testing and debugging issues are outlined, followed by some conclusions and directions for anticipated future progress in distributed software engineering.

2 SPECIFYING DISTRIBUTED SOFTWARE

Specification languages for distributed systems are aimed at describing requirements on the behavior of these systems. In particular, these languages must be able to capture key aspects of distributed software including concurrent process execution, interprocess communication and synchronization, and nondeterministic control. Various mechanisms and tools have been proposed for modeling and specifying distributed software. Here, three general categories, which are based on automata theory, temporal logic, and process algebra, are discussed. These categories have been chosen since they serve as the foundations for many of the key specification methods advocated for distributed and concurrent software. Also, each category represents a different type of modeling and specification notation.

2.1 Automata Theory

Specification methods based on automata theory use notations and concepts from formal language theory. Formal notations for both language recognition (e.g., finite state automata) and language generation (e.g., regular expressions) can be used as a basis for modeling relevant aspects of the behavior of distributed systems. The underlying behavior of the system is defined in terms of event strings, where each string represents one possible sequence of events associated with the system's execution. Language theory and automata theory can then be used to characterize a set of legal event strings, which correspond to the desired system behaviors.

One way that a language-theoretic notation can be used as a specification model is to let symbols for a grammar or regular expresssion correspond to basic events or actions associated with some process. The behavior of a system then corresponds to a set of "legal" interleavings of these symbols. An example of such an approach is the constrained expressions model [Avr91], which is based on an extended form of regular expression notation.

Language theory is also used for defining automata-based methods. Two examples of automata-based methods that have been applied specifically to distributed software are the labeled transition system (LTS) model [Mag99] and the Petri net model [Mur89] [Pet81]. The Petri net model will be described briefly here since it is a very general model with explicit support for the key distributed computing features of parallelism and nondeterminism.

Various kinds of Petri nets have been defined to date. In an ordinary Petri net there are two kinds of nodes, namely place nodes and transition nodes; directed arcs connect from places to transitions and vice versa. A network state associates a nonnegative number of tokens (i.e., markers) with each place. Formally speaking, an ordinary Petri net N is a 4-tuple $N = (P, T, F, M_0)$, where P is a finite set of places, T is a finite set of transitions, F is an arc set, and M_0 is an initial marking, that is, an initial assignment of tokens to each place $p \in P$. Given an arc f from a place p (a transition t) to a transition t (a place p), p is said to be an input (output) place for t and t is said to be an output (input) transition for p.

State changes are carried out by firing enabled transitions. In an ordinary net a transition is enabled when all its input places have at least one token. When an enabled transition t is fired, a token is removed from each input place of t and a token is added to each output place of t; this gives a new state. A net marking is dead when there are no enabled transitions. A Petri net is safe if the number of tokens in each place is either zero or one in the initial state and in every state reachable by firing sequences of enabled transitions.

Applications of Petri nets to concurrent programs have received considerable attention. Two main benefits of Petri nets are their abilities to capture in a natural way the key aspects of distributed software and to support various kinds of analysis techniques, such as simulation and reachability analysis. In this section the modeling capabilities of Petri nets are illustrated by using the well-known producers and consumers example. Analysis issues are discussed later.

A Petri net representing a simple version of the producers and consumers example is shown in Figure 1. In this version a producer process indefinitely iterates the actions of producing a message and depositing the message in a buffer. When a message is available in the buffer, a consumer process can take the message from the buffer and then perform some internal actions.

This example is modeled by an ordinary Petri net with 5 places and 4 transitions. The producer process is modeled by places 1 and 2, and by transitions a and b. The first transition represents the action of producing a message, and the second transition models the action of depositing the message in the buffer. The consumer process is modeled by places 4 and 5, and by transitions c and d. Transition c represents the action of taking a message from the buffer, and transition d

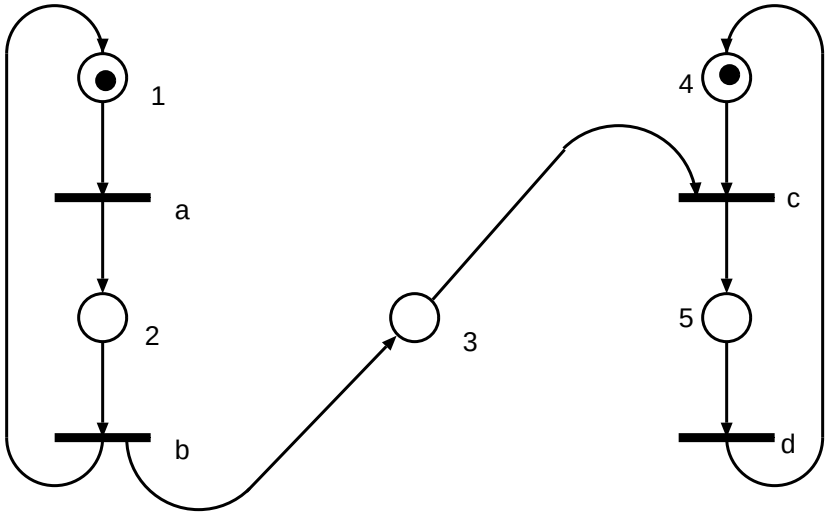

Figure 1: Petri net representation of producers and consumers example

captures the internal actions performed by the consumer. Tokens in place 3 represent messages that have been deposited by the producer but not taken by the consumer.

In the initial marking, places 1 and 4 have one token and the remaining places have no tokens. Only transition a is enabled. When this transition fires the token in place 1 moves to place 2, thus enabling transition b. When b fires, the token in place 2 is removed and a token is added to places 1 and 3. Transitions a and c are now enabled. Firing a puts another token in place 2. When c fires a token is removed from places 3 and 4, and a token is added to place 5, thus enabling transition d. When this transition is fired, a token is returned to place 4 from place 5. Note that in this Petri net the size of the buffer is unbounded. Of course, different ordinary Petri nets can be constructed that model bounded buffers as well as multiple producer and consumer processes.

Various extensions to ordinary Petri nets have been defined. Some extensions incorporate the concept of time in Petri nets aiming at making the model more suitable for performance modeling and analysis [Ajm95] [Flo91] [Mol85] [Ghe91]. Other extensions are aimed at increasing the expressive power or the modeling convenience of Petri nets. Examples are the introduction of inhibitor arcs in Petri nets and the notion of colored Petri nets. A discussion of these kinds of nets is beyond the scope of this article. See [Mur89] for an overview of different Petri net models.

2.2 Temporal Logic

Temporal logic is an extension of classical logic for dealing with systems that evolve with time. In particular, temporal logic formulas usually relate the value of system variables in a given state

to variable values in subsequent states. An advantage of this approach is that temporal logic can capture in a concise and uniform way most safety and liveness properites of distributed systems. Following Lamport [Lam80], a safety property is a property that must hold continuously during system execution, and a liveness property is a property that must hold at given times during system execution. In particular, safety properties typically express that "something bad should never happen," whereas liveness properties express that "something good should eventually happen." Freedom from deadlock and the compliance with mutual exclusion constraints are examples of safety properties, whereas the requirement that a request for a given service (e.g., the allocation of a resource) be eventually satisfied is an example of a liveness property.

An additional advantage of temporal logic specifications is that they support various kinds of analysis techniques. These techniques generally fall into two categories: Proof-theoretic reasoning and model-checking. A discussion of these analysis techniques is beyond the scope of this article. The interested reader is referred to [Kro87] [Man92] [Man92a].

The sequential model underlying linear-time temporal logic contrasts with the branching-time model. In the former model the elapsing of time is viewed as a sequence of transitions from one state to a unique subsequent state. In the latter model the evolution of time is viewed as a continuous branching among many possible future states. Consequently, the linear-time model is a totally ordered sequence of states, whereas the branching-time model consists of a tree of partially ordered states. These two approaches have been analyzed and compared extensively [Eme83] [Lam80]. These investigations have shown that branching-time logic can provide more expressive power than linear-time logic. In many cases, however, the expressive power and the reasoning capabilities provided by linear-time logic are sufficient for the application at hand. The example given below uses a simple linear-time propositional temporal logic (PTL).

In linear-time PTL, time is viewed as a sequence of states in which each state is characterized by the assignment of a truth value to a set of propositional variables. Both the standard propositional connectives and temporal operators can be used in PTL formulas. The former ones include, for instance, $\wedge$ (and), $\vee$ (or), $\neg$ (not), and $\rightarrow$ (implies), with the usual meanings. The latter ones typically include the following (assuming that p and q are logic formulas):

$\Box p$	(**always** p)	p true at current state and any subsequent state.
$\Diamond p$	(**eventually** p)	Either p true at current state or in some future state.
$\circ p$	(**next** p)	p true at state immediately following current state.

$p \mathcal{U} q$ (p **until** q) Either q true at current state or p true at least as long as q false.

Note that this set of temporal operators is not primitive, in that some operators can be expressed in terms of others. For instance, the eventually operator can be expressed in terms of the always operator because of the following equivalence:

$$\Diamond p \equiv \neg \Box \neg p$$

This formula asserts that p must eventually be true if and only if p is not always false.

As an example of a temporal logic specification, consider the semaphore specification described in [Kar84]. The goal is to express the requirements of a binary semaphore, which provides the two standard operations P and V. Assume that V always sets the semaphore variable to 1. The following notation is used:
$a \in P$: A process a has called operation P and is waiting for the operation to complete.
$a \in V$: A process a has called operation V and is waiting for the operation to complete.
s: The value of the binary semaphore variable (either 0 or 1).

Here are two temporal logic formulas expressing a safety property and a liveness property for the semaphore. Note that additional temporal logic formulas are required for a full specification. Those formulas can be found in [Kar84].

1. A P operation cannot complete if the semaphore variable is 0 (i.e., the semaphore is unavailable) and the value never changes:

$$(a \in P \wedge \Box(s = 0)) \rightarrow \Box(a \in P)$$

2. A process that has called the P operation will not be blocked indefinitely if the semaphore value becomes 1 infinitely often (i.e., if there is no state after which the semaphore is always 0). This liveness property is typically satisfied by implementing a semaphore with a first-in-first-out queue:

$$((a \in P) \wedge \Box\Diamond(s = 1)) \rightarrow \Diamond(a \notin P)$$

2.3 Process Algebra

A third class of specification models is the algebraic model. The foundational works in this area are Milner's CCS [Mil89] (Calculus of Communicating Systems) and Hoare's CSP [Hoa85] (Communicating Sequential Processes). Both of these notations provide algebraic theories for describing systems composed of computing elements (agents) that communicate over channels. Such models

are called process algebra models [Bea90]. A major feature of process algebra is that it gives sound mathematical semantics to the all-important concepts of process, concurrency, nondeterminism and communication. Furthermore, these algebraic theories provide a framework for defining complex behaviors from simpler behaviors by use of such operators as concatenation and parallel composition. Given process algebra-based notations that describe the behavior of processes, it is possible to then formally define forms of equivalance. In a simple sense, two processes are equivalant if their executions correspond to identical event sequences. The algebra will typicially support the notion that some events are not externally observable. Thus, equivalence depends not only on the process definitions, but also on the specific types of events that are of concern. Some theories, such as CCS and CCS-based theories, support this concept by providing features (e.g., restriction and hiding operators) that support modularity and abstraction in program specifications.

The development of the language LOTOS, which is based on CCS, is one of the earliest steps toward a practical application of process algebras to software engineering [Lot89]. More recent efforts in applying process algebra include the development of methods and tools like the Concurrency Workbench [CPS93] and the PARAGON toolset [BACLS97].

3 DESIGN ISSUES

Two important design issues underlying distributed systems are the definition of the processes in a system and the allocation of processes to physical processors. Of course, these design choices are strongly dependent on the nature of the application and the configuration of the processing environment. On the one hand, for many real-time control systems operating on a dedicated network (e.g., navigation control in aircraft), the system designer has lots of flexibility in defining processes and their allocation. On the other hand, the design of many Internet-based applications is driven by the ability to dynamically move applets to various client machines on demand. This implies a much more restricted form of distributed computing and a limited concern at design time for process allocation. The remainder of this discussion considers the case in which it is necessary at design time to identify processes and their allocation to processing resources (or at least an initial allocation that might well change during execution in response to handling of load changes and/or faults).

As with traditional software design, designers use the requirements of a distributed system to identify the logical modules and the main data structures in the system. Traditional design methodologies, such as functional decomposition and object-oriented design, can also be applied

to the definition of the logical structure of distributed software. In fact, structuring of distributed systems as distributed objects has become a much advocated design approach. In the case of distributed systems, however, the logical modules and data structures so obtained are generally viewed as the basis for the definition of asynchronous processes contained in the system. This activity is usually called task partitioning. The activity of task allocation then assigns the processes identified during task partitioning to physical processors that execute the corresponding processes. In the next two subsections both of these activities are discussed.

3.1 Task Partitioning

Task partitioning is the mapping of logical modules and data, which reflect the application's point of view, into a set of actual processes and data files, which reflect the designer's point of view. Here, module partitioning, rather than data partitioning, is considered. For any system, there are many ways we can partition the logical modules in the system into a set of processes. For example, it is possible to consider putting all modules in a single process; or to randomly assign each module to one of some fixed number of processes. Task partitioning is important because partitioning defines processes, and processes are the major program structures for distributed programs.

The objectives of task partitioning must support design goals, such as minimization of completion time, achieving load balance, maximizing reliability, or improving capability for system growth. Three typical examples of task partitioning objectives are (1) exploiting potential concurrency, (2) minimizing interprocess communication, and (3) limiting the size of processes. (A partition consisting of many small processes provides greater flexibility than a partition consisting of a few large processes in the subsequent task allocation step.)

The activity of task partitioning is often difficult because different design objectives can lead to conflicting design decisions. For instance, while the goals of achieving parallelism and limiting process size lead to designs with many, relatively small processes, the goal of minimizing interprocess communication leads to few, large processes. Consequently, software designers are often required to make tradeoffs among various criteria in order to reach an effective partitioning for use in the allocation stage.

An additional difficulty is that in many cases the effectiveness of a task partition cannot be fully assessed before the subsequent step of task allocation. For this reason task partitioning and allocation are often iterated in an effort to provide mutual feedback between these two activities. Finally, task partitioning must take into account relevant constraints, such as precedence relations between processes, timing requirements, and limited capacities of different resources (e.g., CPU,

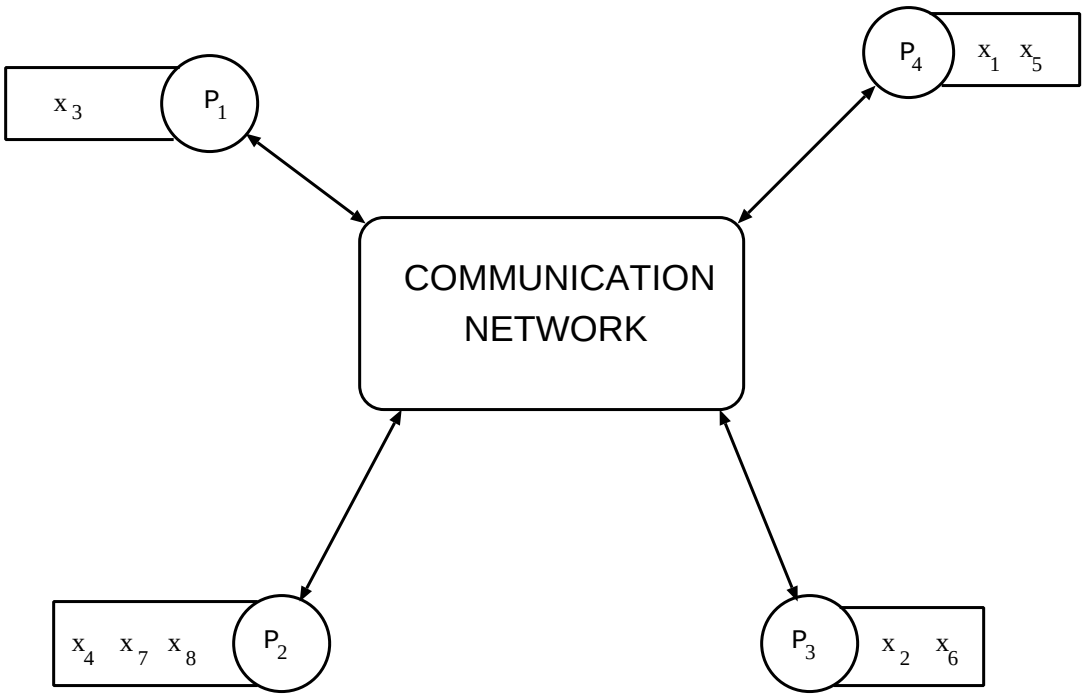

Figure 2: Example of a task allocation

memory, and communication link bandwidth).

3.2 Task Allocation

Task allocation assigns each process and data file defined during partitioning to one or more system processors. (The reason for assigning a process or data file to multiple processors is to help meet fault-tolerance or performanced requirements). The allocation is said to be static if it is performed before system execution, and dynamic if it is performed during execution. Dynamic allocation is often used in distributed or network operating systems to support load balancing or fault recovery. Here the discussion is limited to the basic ideas and issues related to static allocation. More complex issues, and associated methods, arise in practice, where features like real-time constraints become important (see for example [Pen97]).

As an example, Figure 2 illustrates a four processor system with the following static allocation of 8 processes $x_1 \ldots x_8$: Processor P_1 executes x_3; processor P_2 executes x_4, x_7, and x_8; processor P_3 executes x_2 and x_6; and processor P_4 executes x_1 and x_5.

Two relevant issues underly static task allocation. First, suitable metrics must be defined to assess the effectiveness of each allocation. The second issue is concerned with the definition of methods for finding "good" allocations that also satisfy constraints on the virtual machine (in

general both hardware and software) on which the system is executed. An example of such a constraint is an upper bound on the number of processes that can be executed on a given processor due to limited memory capacity. First allocation metrics are discussed and then algorithms for finding good task allocations are mentioned.

Various metrics have been defined for evaluating task allocations. These metrics usually associate a given cost with each allocation. Of course, different metrics use different cost functions for determining the cost of an allocation. Here the following four metrics are introduced: (1) IPC (interprocess communication); (2) E+IPC (execution time plus interprocess communication); (3) CT (completion time); and (4) LB (load balancing). It should be noted that most of the work on defining and evaluating allocation metrics tends to be theoretical. In practice, it is difficult to obtain the type of data that is needed to put the theory into practice. Also, optimal task allocation has an inherent high computation cost. This has motivated investigation and use of heuristic approaches to this problem.

In the IPC metric the cost of an allocation is determined only by the amount of interprocess communication during system execution. This metric is based on the assumption that the time required by communication is predominant with respect to process execution time. Because the cost of a communication between two processes is usually higher if the processes are on different processors, this metric tends to assign a lower cost to allocations in which many processes are placed on few processors. (The communication cost is actually minimal if all processes are on a given processor; however, this kind of allocation may violate some implementation constraint.)

A disadvantage of the IPC metric is that it does not consider the effects of process execution time. In addition, when processors differ in their execution speed or in the availability of resources (e.g., hardware for floating point computations), the impact of the different processors on the execution time of each process is not taken into account.

The E+IPC metric obviates the above disadvantages of the IPC metric. In the E+IPC metric the cost of a given allocation is given by the sum of process execution time and interprocess communication time. Moreover, the execution time of a given process P depends on the processor to which P is assigned. Thus, the E+IPC metric quantifies the total resource usage of a distributed system, where the resources considered are processors and communication links. When the system consists of a single thread of execution (i.e., when processes must be executed in a sequence), the E+IPC metric can also model the time required to execute the system. A disadvantage, however, is that this metric fails to take into account the effects of parallel execution on system performance. This disadvantage is sidestepped by the completion time (CT) metric.

The CT metric is computed first by finding the E+IPC cost of each processor, and then by using the processor whose E+IPC cost is highest. This metric accounts for the fact that the executions of the processors overlap in time but does not consider the fact that a processor can sometimes be idle because all processes assigned to it are waiting for some synchronization event to take place.

Intuitively the notion of load balance, which underlies the LB metric, implies the desire to spread the load as evenly as possible among the host processors. A cost metric for load balance must precisely define the load value. For example, a very simple metric for load balance might define load as the number of processes allocated to a processor. In this case, an optimal task allocation algorithm would try to have each processor execute the same number of processes, regardless of the amount of time that each processor spends in its execution. This type of load balancing may be well-suited for network operating systems that must dynamically monitor the job queues of the host processors; however, for distributed computing it is more appropriate to consider a processor's load to be the sum of the execution times associated with the processes allocated to that processor. In this case the load balancing refers to a leveling of CPU use or processor utilization.

Now consider the problem of finding a good task allocation for a given system. In general there are three basic methods for solving this problem: optimal methods, approximation methods, and probabilistic methods. Optimal methods yield optimal results; however, they have high computational complexity. Approximation methods often avoid the computational intractability of optimal methods but generally produce suboptimal solutions. Both optimal and approximation methods use exact estimates of process execution times and of interprocess communication costs in a distributed system; however, such estimates often cannot be easily obtained. Probabilistic methods use random variables to characterize execution times and communication costs. The problem of finding exact estimates is therefore circumvented by these methods. Note that, except for systems with very few processes and processors, the number of possible allocations is usually very large. Even without considering the possibility of processes being assigned to multiple processors, there are m^n possible allocations of n processes on m processors.

As an example of an optimal method, consider the problem of finding the allocation with minimum E+IPC cost. It has been shows that this problem can be cast as an integer programming problem and also as a graph theory problem. This is significant since it demonstrates that formal representations of a problem allow for the use of existing theory and algorithms to define solutions. One challenge in formalizing the task allocation problem is in handling of allocation constraints. The integer programming approach allows many constraints on task allocation to be expressed naturally. For instance, the constraint that the sum of the memory requirements of all processes

assigned to a given processor k must not exceed the memory available on processor k can be expressed with the following linear inequality:

$$\sum_{i=1}^{m} s_i x_{i,k} \leq r_k \tag{1}$$

where m is the number of processes, s_i is the memory requirement of process i, $x_{i,k}$ is an allocation variable (boolean) that has value 1 if module i is allocated to processor k, and r_k is the memory available on processor k. Thus, using the integer programming approach, the problem of finding an optimal task allocation is solved by minimizing a given cost function, subject to a set of constraints such as (1) above. Additional approaches to optimal task allocation have been defined. See, for instance, Stone's graph-based method for minimizing E+IPC cost on a two processor machine [Sto77]. An approximation method for task allocation is given in [Efe82], and probabilistic methods are described in [Ind86] and [Nic89].

4 PROGRAMMING LANGUAGE ISSUES

There are two basic approaches to writing code for distributed systems. The first approach is to use a traditional (sequential) programming language to implement each process in the system. Calls to operating system functions are used for interprocess communication and synchronization. The second approach is to use a concurrent programming language. Languages of this kind explicitly define constructs for specifying units that can be executed in parallel (e.g., processes) and constructs that allow such units to communicate with each other.

The second approach—using a concurrent language—has several advantages with respect to the first one. First of all, the flexibility provided by the interprocess communication constructs of a concurrent language in general cannot be matched by operating system defined primitives. In particular, concurrent languages allow a process to wait simultaneously on multiple possible communications and to choose nondeterministically among such communications. Moreover, concurrent languages usually allow complex data structures, such as arrays and records, to be passed between asynchronous processes, unlike operating system primitives. An additional advantage of concurrent languages is that they enhance portability. Finally, concurrent languages are beneficial to software reliability because (1) they support static type checking and (2) they are more amenable to the development of design and analysis tools for distributed software than sequential languages.

A variety of languages for programming distributed systems has been defined. Examples of such languages include Ada 83 [Uni83], Ada 95 [Bar96], Argus [Lis88], Concurrent C [Geh89],

Concurrent Smalltalk [Yok87], CSP [Hoa85], Java [AGH00, Lea97], Parlog [Cla86], Linda [Ahu88], NIL [Str86], Occam [Occ84], ParAlfl [Hud88], and SR [And88]. Most of these languages are used as research vehicles; however, a few of them are also commercially available (e.g., Ada 83 and 95, Concurrent C, Java, and Occam). Many detailed surveys of language issues for distributed systems exist. See, for instance, [Bal89, BGL98, ST98, Tho87, Tho97]. Here, the most important issues are summarized: (1) Granularity of parallelism; (2) synchronous vs. asynchronous communication; (3) distributed vs. shared address space; and (4) handling of error conditions.

Granularity of parallelism. The granularity of parallelism supported by a concurrent language refers to the size of the units that can be executed in parallel. On the one hand, languages supporting large-grained parallelism allow for the definition of sizable parallel units. Examples of such units are Ada's tasks, Concurrent C's processes, and Concurrent Smalltalk's active objects. On the other hand, languages supporting fine-grained parallelism allow statements or even portions of statements to be executed in parallel. For instance, functional languages for distributed systems – such as ParAlfl – allow arguments of function invocations to be evaluated in parallel. Likewise, relational languages – such as Parlog – support two kinds of parallelism (i.e., AND parallelism and OR parallelism) in the evaluation of the clauses appearing in a program.

A potential advantage of fine-grained parallelism is that a suitable language system, such as a parallelizing compiler, can automatically detect portions of a program that can be executed in parallel. Thus, in theory a programmer need not perform the activities of task partitioning and allocation. In practice, however, the overhead from process creation and interprocess communication is likely to offset the advantages of parallelism if the units that are executed in parallel are too small. Consequently, even languages supporting fine-grained parallelism often allow programmers to control the units that can be executed in parallel [Hud88].

Synchronous vs. asynchronous communication. In synchronous communication two processes must reach predetermined communication points in their flow of control in order to exchange information with each other. Thus, if a process A reaches a communication point before B reaches the corresponding communication point, A must wait for B and vice versa. In the asynchronous case, a process A can send data to a process B and continue its execution without waiting for B to be ready to receive such data. In general, this type of communication requires the definition of suitable structures, such as channels or ports, whose purpose is to hold data that was sent but not yet received. Note that, even in the case of asynchronous communication, a process that is ready to receive data may be delayed if no data has been sent.

An obvious advantage of asynchronous communication is that it leads to greater parallelism

than synchronous communication because in general a message sender need not wait until the receiver has processed the message. In addition, asynchronous communication is more flexible than synchronous communication because the effects of the latter can be achieved in a relatively straightforward way with asynchronous constructs, whereas the opposite (i.e., achieving the effects of asynchronous communication with synchronous constructs) is not as easy. Suppose that process A must synchronize with process B using asynchronous communication. This can be accomplished with two asynchronous sends, one from A to B, the other from B to A. In this way, both A and B must mutually wait for each other. Now suppose that the effects of an asynchronous send must be achieved by use of synchronous communication. In general, this requires the definition of an additional process C acting as a buffer between A and B.

A disadvantage of asynchronous communication is that it is generally more difficult to implement than the synchronous case because channels or ports are needed. In addition, the behavior of a communication must be suitably defined when a channel or a port overflows. One possibility is to allow data in the channel or port to be overwritten. This behavior can be error-prone, however, because it can cause data to be lost. Another possibility is to force a sending process to be delayed until the intended receiver has taken some data from the channel or port. In this case, however, a process sending data could be delayed just as with synchronous communication. Some commercially available languages, such as Ada 83 and Occam, use synchronous communication. NIL is an example of a language supporting asynchronous communication. Ada 95 supports asynchronous communication through protected objects and synchronous communication through the rendezvous mechanism, which was originally defined in Ada 83, and remote procedure calls. The Java language supports asynchronous communication through monitors [Bri78] and remote method invocations (RMIs) [AGH00]; Java also supports asynchronous communication through sockets and byte streams. Finally, Concurrent C and SR also support both kinds of communication.

Distributed vs. shared address spaces. In the case of a distributed address space, each parallel unit (e.g., process) has access to its own address space and all interprocess communication occurs by message passing. In shared address spaces, multiple parallel units can access and modify common sets of data structures. Of course, the shared data structures need not be implemented on the same physical medium but are often implemented in a distributed fashion.

Most languages for distributed systems use distributed address spaces (e.g., Ada 83, Argus, Concurrent C, CSP, Occam, SR). Java and Ada 95 support shared address spaces through monitors and protected objects, respectively. Functional and relational languages typically use shared logical variables, which are a form of shared address space. These variables satisfy the "single assignment

property," that is, after these variables are assigned a value, they are never modified. Distributed processes can communicate and synchronize through such variables. For instance, a process A that must read a shared variable x is delayed if a value has not been assigned to x.

A different form of shared address space is used in the language Linda. In this language, processes communicate via a structure called a tuple space. The tuple space is an ordered collection of tuples of various types and sizes. Access to tuples is associative, that is, a process can read the content of a tuple by supplying a part of its contents. If the tuple space contains no tuple that matches the portion specified by a process, the process is blocked. The operation of reading a tuple can be destructive. In this case the tuple matching a process specification is removed from the tuple space. Of course, processes can also add tuples to the tuple space. For instance, an integer global variable could be represented by a tuple consisting of two elements, a string denoting the variable name and an integer denoting the variable value: ["x", 3]. A process would then use the following two statements to increase the value of the variable:

```
in("x", var i)
out("x", i+1)
```

The first statement destructively reads the tuple whose first element matches "x" and assigns the second element of this tuple to variable i. The second statement adds a tuple whose elements are "x" and the value of the expression $(i + 1)$.

A significant benefit of a tuple space that it uncouples processes from each other in both time and space. Processes are uncoupled in time since they need not exist at the same time to communicate with each other. In fact, a process A may create some data tuples and then terminate. Subsequently, some new processes could be created that read the tuples produced by A. Processes are uncoupled in space since a process need not name a partner in a communication, unlike most other languages. A disadvantage of the Linda approach is that a distributed implementation of a tuple space requires fairly sophisticated protocols, especially when tuples are replicated across different processors for performance or fault-tolerance reasons. Recently, Picco et al. have suggested that Linda's approach to specifying interprocess communication is well-suited to modeling mobile computing applications [PMR99].

Handling of error conditions. These conditions may be caused by software flaws or by hardware failures, such as a processor or a disk crash. Various languages take different approaches to dealing with error conditions. At one extreme there are languages that do not provide any support for dealing with these conditions (e.g., Occam, DP, and Emerald). At the opposite extreme

there are languages, such as NIL, that handle processor failures transparently to the programmer by means of sophisticated recovery techniques.

Some languages take an intermediate approach. For instance, Ada 95 automatically detects certain software errors, such as a process (or task in Ada 95 terminology) trying to establish a communication with a process that has already terminated its execution. A similar mechanism detects certain error conditions during RMI calls in Java [AGH00]. The detection of such conditions in Ada and Java plugs nicely into a language-defined exception handling mechanism, a set of language constructs for dealing with a wide variety of error conditions (i.e., conditions not necessarily related to interprocess communication). Argus provides some support for fault-tolerant computing essentially by use of the notion of atomic execution: Either all or none of the results of a given computation are carried out depending on whether a hardware failure occurs. In addition, Argus supports the definition of so called stable data structures. An updated copy of stable structures is kept in permanent storage (e.g., on a disk), thus allowing the state of these structures to be recovered in the event of a processor failure.

In general, the ability to deal with error conditions can be beneficial especially in the case of critical applications with fault-tolerance requirements; however, the added overhead of transparent error recovery may not be justified for applications that have no such requirements.

This completes the discussion of the four language issues identified earlier. Note, however, that there are several additional issues that cannot be discussed here because of space limitations. These issues include: (1) Supporting broadcast as opposed to point-to-point communication; (2) allowing dynamic process creation instead of forcing a fixed process structure defined at compile-time; (3) allowing two-way vs. one-way flow of data during a single point-to-point communication; (4) requiring a process to name explicitly its communication partners; and (5) providing constructs for controlling nondeterministic choices among many possible communications within each process. See [Bal89], [BGL98], [ST98], [Tho87], and [Tho97] for a discussion of these issues.

5 TESTING AND ANALYSIS

The activities of software testing and debugging are generally known to be difficult and error prone even for traditional (sequential) programs. Additional difficulties arise in the case of distributed systems because process computations in these settings can be interleaved arbitrarily and because system behavior is often nondeterministic. Furthermore, traditional testing and simulation techniques are often impractical or inadequate because the number of interleavings of computations

performed by asynchronous processes is usually very large. These facts make the definition of formal methods for static analysis of distributed software an attractive alternative to traditional testing and debugging techniques. Of course, in order to be helpful to program developers these methods should lead to the development of automated tools for testing and analysis of distributed systems; the behavior of these systems is much too complex to permit effective analyses by hand.

Before discussing any particular analysis or testing approaches, it is important to identify the inherent limitations of such approaches. On the one hand, consider dynamic testing, which is based on actual execution of the software using test cases (i.e., input values). Clearly, the observed behaviors represent only a subset of the possible behaviors of the software system. So, this subset of behaviors may only represent a small fraction of the possible process interactions. This may mean that potentially critical properties, such as deadlocks, or other race conditions, are not observed. On the other hand, it is possible that some of the behaviors observed through static analysis—and the associated faults—are not "true" behaviors. Any observed faults must be further investigated and corrected as part of the debugging process.

Now consider using analysis approaches, meaning approaches based on static analysis. Static analysis methods consider a software system's structure and do not rely on actual executions using test case data. Because this analysis is performed on a static representation of the software (typically using a model derived from the source code), it is primarily control-flow oriented. In general, the analysis must be independent of dynamic features such as message delays and run-time bindings of data values. Consider a process that acts as a print server for a system of 100 client processes. Assume that requests for different classes of print services are to be handled differently depending upon the identity of the requesting client and the order in which the requests are received at the server. Dynamic testing, even when done very carefully, might not happen to activate some particular (and possibly rare to occur) execution sequence that contains a fault. Maybe such a fault occurs when a lower priority client makes a request for some service and this request arrives to the server before a different request from a higher priority client. But, static analysis should evaluate this case, along with all other statically identifiable cases. In this sense, static analysis is more "complete" than dynamic testing. Of course, by considering all statically identifiable source code paths (and ignoring some data-dependent constructs) it is likely that some infeasible paths and associated faults will be detected. These are known as spurious errors. Naturally, it is important to try and minimize the number of spurious errors that are reported during static analysis.

The following is a brief survey of some of the distributed software analysis approaches that have been studied. This is followed by a brief discussion of some dynamic testing techniques.

The goal of static analysis is to determine whether a distributed system has certain desirable properties based on an examination of the system's source code. Typical properties that are the target of static analysis include both safety properties, such as freedom from deadlock and satisfaction of mutual exclusion, and liveness properties, such as absence of starvation.

Existing approaches to analysis of distributed software fall into four main categories. Some approaches are based on a reachability analysis of a model of a distributed system. The goal of this analysis is the construction of the set of possible states of the system (e.g., [Kar90], [Sha96], [Tay83]). Other approaches, such as constrained expressions [Avr91] and the Petri net T-invariant approach [Mur89a], use linear algebra based techniques to answer questions about the properties of a distributed system. Approaches based on the use of testing or simulation techniques are aimed at analyzing executions of a distributed system or simulations of the system [Hel85]. Finally, various approaches are aimed at theorem proving in some logical structure associated with a distributed system [Cla86a], [Kar84], [Owi82], [Che98]. Here, our discussion is limited to reachability-based approaches and dynamic testing-based techniques.

Reachability-based approaches. These approaches are aimed at the construction of the state space of a distributed system. The state space is the set of states that can be reached from the initial state by the asynchronous processes that make up the system. For distributed software, the state space can be constructed starting from a specification or from actual source code. For example, the analysis tool SPIN accepts specifications written in a language called PROMELA (Process Meta Language) [Hol97], while the TOTAL system (Tasking Oriented Toolkit for the Ada Language) creates Petri net models from Ada source [Dur94]. Traditionally, a state space is represented as a "full" reachability graph for the system, with explicit enumeration of all reachable states and their transitions [Tay83]. An alternative approach that has been proposed and investigated is to describe the state space (states and transitions) in terms of boolean formulas which can be encoded efficiently in a special data structure like a binary decision diagram (BDD). Analysis of such a symbolic (or implicit) representation of the state sapce is commonly called symbolic model checking [McM93]. This approach has been also used in the development of HyTech [ACH+95, Hen97], a tool for analysis of hybrid systems, systems that involve both discrete-time and continuous-time components. One commonly considered analysis goal of reachability techniques is to determine whether a program is free from deadlock. Some experimental results for deadlock detection using reachability techniques can be found in [Dur94] and [Cor96].

A main disadvantage of reachability-based approaches is the "state explosion problem." This refers to the fact that the size of the program's state space is frequently exponential in the number

of processes in a distributed system [Tay83a]. Much of the active research associated with static analysis for distributed software is aimed at development of methods to reduce the computational complexity inherent in reachability-based techniques. In [Sha96] a technique is described whose goal is to reduce the size of the Petri net model of a distributed system. The reduction in the model size usually results in a very substantial reduction in the size of the corresponding state space. Other techniques exploit the fact that computations performed by different processes are often independent of each other, thus avoiding the generation of all possible interleavings of such computations [God91] [Val90]. Finally, some techniques take advantage of symmetries resulting from asynchronous processes with similar behavior in order to reduce the size of a state space [McD89] [Sta91].

Dynamic testing. Dynamic testing and debugging of distributed systems are complicated by the following factors. First, the nondeterministic behavior of these systems makes it difficult to reproduce program behaviors. Second, because distributed systems are usually executed on multiple processors, dynamic testing and debugging often require observing events and collecting data from different processors. Several techniques have been defined to address the above difficulties. A common disadvantage of any dynamic testing technique is that in general it is difficult to force dynamically all possible behaviors of a distributed system due to their very large number.

Event debugging is the collection and storage of event information during the execution of a distributed system. Events can be recorded either by suitable statements inserted in the system's source code, or by logging capabilities incorporated in the virtual machine that executes the system. A variation of this technique is event monitoring. In event monitoring the debugger not only records events in execution logs, but also evaluates the events in order to check for the occurrence of certain conditions, such as deadlock [Hel85].

An alternative testing and debugging technique is execution control [Tai91]. The goal of this technique is to force a desired sequence of events during the execution of a distributed system. For instance, a programmer can use execution control to reproduce a previously observed sequence of events [Wit88]. (This approach is sometimes called execution replay.) A survey of this and other techniques for debugging distributed software can be found in [McD89a]. An approach to generate test cases based on the specification of event constraints is described in [Car98].

6 CONCLUSIONS

In this article the main issues characterizing the production of reliable distributed software systems were surveyed. In particular, the unique issues that must be considered in the development of distributed software were discussed, and the effects of distribution on the various phases of the software lifecycle were analyzed.

The development of distributed systems stands to benefit significantly from past and present research efforts. For instance, the adoption of concurrent languages – as opposed to sequential languages with calls to operating system primitives – will clearly enhance the portability and the readability of distributed software. Moreover, formal specification and analysis techniques are likely to be incorporated in software development environments, thus enhancing the reliability of the resulting software and reducing production costs. In particular, the use of formal specification and modeling tools is likely to give software designers and implementors a better understanding of the behavior of a distributed system during system design. Consequently, potential design flaws will be detected early in the software lifecycle, thus reducing the cost of correcting such flaws. Finally, automated analysis techniques will help developers detect undesirable situations, such as deadlock and starvation, more quickly and more thoroughly than simulation techniques, which have frequently been used for practical development of distributed software.

References

[Ahu88] S. Ahuja, N. Carriero, and D. Gelernter, "Matching langauge and hardware for parallel computation in the Linda machine," *IEEE Trans. Computers,* 37(8):921–929, August 1988.

[Ajm95] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modeling with Generalized Stochastic Petri nets.* John Wiley & Sons, Inc., New York, New York, 1995.

[ACH+95] R. Alur, C. Courcoubetis, T. A. Henzinger P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, February 1995.

[And88] G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend, "An overview of the SR language and implementation," *ACM Trans. Progr. Lang. Syst.*, 10(1):51–86, Jan. 1988.

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, third edition, 2000.

[Ath88] W. C. Athas and C. Seitz, "Multicomputers: Message passing concurrent computers," *IEEE Computer*, 21(8):9–24, Aug. 1988.

[Avr91] G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden, "Automated analysis of concurrent systems with the constrained expression toolset," *IEEE Trans. Software Engineering*, 17(11):1204–1222, Nov. 1991.

[Bal89] H. E. Bal, J .G. Steiner, and A. S. Tanenbaum, "Programming languages for distributed computing systems," *ACM Computing Surveys*, 21(3):261–322, Sept. 1989.

[Bar96] John Barnes. *Programming in Ada 95.* Addison-Wesley, Reading, Mass., 1996.

[Bea90] J. C. M. Beaten and W. P. Weijland, *Process algebra,* Series on Cambridge Tracts in Theoretical Computer Science, Vol. 18, Cambridge University Press, Cambridge, England, 1990.

[BACLS97] Hanêne Ben-Abdallah, Duncan Clarke, Insup Lee, and Oleg Sokolsky. PARAGON: A paradigm for the specification, verification, and testing of real-time systems. In *Proceedings of the IEEE Aerospace Conference*, pages 469–488. IEEE Computer Society, February 1997.

[Bla87] A. P. Black, N. C. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distribution and abstract types in Emerald," *IEEE Trans. Software Engineering,* 13(1):65–76, Jan. 1987.

[Bri78] P. Brinch Hansen, "Distributed processes: A concurrent programming concept," *Comm. ACM,* 21(11):934–941, Nov. 1978.

[BGL98] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.

[Car98] R. Carver and K. C. Tai, "Use of sequencing constraints for specification-based testing of concurrent programs," *IEEE Trans. Software Engineering,* 24(6):471–490, June 1998.

[Che98] B. Chetali, "Formal Verification of Concurrent Programs Using the Larch Prover," *IEEE Trans. Software Engineering*, 24(1):46–62, Jan. 1998.

[Cla86] K. L. Clark and S. Gregory, "Parlog: Parallel programming in logic," *ACM Trans. Progr. Lang. Syst.*, 8(1):1–49, Jan. 1986.

[Cla86a] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Progr. Lang. Syst.*, 8(2):244–263, April 1986.

[CPS93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[Cor96] J. C. Corbett, "Evaluating deadlock detection methods for concurrent software," *IEEE Trans. Softw. Eng.*, 22(3):161–179, Mar. 1996.

[Dur94] S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz, "Application and experimental evaluation of state space reduction methods for deadlock analysis in Ada," *ACM Trans. Software Engineering and Methodology*, 3(4):340–380, Oct. 1994.

[Efe82] K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *IEEE Computer,* 15(6):50–56, June 1982.

[Eme83] E. A. Emerson and J. Y. Halpern, "'Sometimes' and 'not never' revisited: on branching versus linear time," *Proceedings 10th Symp. on Principles of Programming Languages,* 1983, 127–140.

[Flo91] G. Florin, C. Fraize, and S. Natkin, "Stochastic Petri nets: Properties, applications and tools," *Microelectronics Reliability,* 31(4):669–697, Pergamon Press, April 1991.

[Geh89] N. H. Gehani and W. D. Roome, *The Concurrent C programming language,* Silicon Press, Summit, New Jersey, 1989.

[Ghe91] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè, "A unified high-level Petri net formalism for time-critical systems," *IEEE Trans. Software Engineering,* 17(2):160–172, Feb. 1991.

[God91] P. Godefroid, "Using partial orders for the efficient verification of deadlock freedom and safety properties," in K. G. Larsen and A. Skou, editors, *Proc. 3rd Workshop on Computer-Aided Verification*, LNCS 575, Springer-Verlag, New York, New York, pp. 332–342, 1991.

[Hel85] D. Helmbold and D. Luckham, "Debugging Ada tasking programs," *IEEE Software*, 2(2):47–57, March 1985.

[Hen97] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, "HyTech: A model checker for hybrid systems," *Software Tools for Technology Transfer*, 1997.

[Hoa85] C. A. R. Hoare, *Communicating sequential processes,* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[Hol97] Gerald J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[Hud88] P. Hudak, "Exploring parafunctional programming: Separating the what from the how," *IEEE Software,* 5(1):54–61, Jan. 1988.

[Ind86] B. Indurkhya, H. S. Stone, and Xi-Cheng Lu, "Optimal partitioning of randomly generated distributed programs," *IEEE Trans. Software Engineering*, 12(3):483–495, March 1986.

[Occ84] INMOS Ltd., *Occam programming manual,* Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Lot89] ISO, *Lotos, a formal description technique based on the temporal ordering of observational behaviour,* International Standard ISO 8807, 1989.

[Kar90] G. M. Karam and R. J. Buhr, "Starvation and critical race analyzers for Ada," *IEEE Trans. Software Engineering*, 16(8):829–843, Aug. 1990.

[Kar84] R. A. Karp, "Proving failure-free properties of concurrent systems using temporal logic," *ACM Trans. Prog. Lang. Syst.*, 6(2):239–253, April 1984.

[Kro87] F. Kröger, *Temporal Logic of Programs*, Springer-Verlag, New York, New York, 1987.

[Lam80] L. Lamport, "'Sometime' is sometimes 'not never'," *Proceedings 7th Symp. on Principles of Programming Languages*, Jan. 1980, 174–185.

[Lea97] Doug Lea. *Concurrent Programming in Java.* Addison-Wesley, Reading, Massachusetts, 1997.

[Lis88] B. Liskov, "Distributed programming in Argus," *Comm. ACM,* 31(3):300-312, March 1988.

[Mag99] J. Magee and J. Kramer, *Concurrency – State models and Java programs,* John Wiley & Sons, Inc., New York, New York, 1999.

[Man92] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems (specification),* Springer-Verlag, New York, New York, 1992.

[Man92a] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems (verification),* Springer-Verlag, New York, New York, 1992.

[McD89] C. E. McDowell, "A practical algorithm for static analysis of parallel programs," *J. Parallel and Distributed Process.*, 6:515–536, June 1989.

[McD89a] C. E. McDowell and D. P. Helmbold, "Debugging concurrent programs," *ACM Computing Surveys,* 21(4):593-622, Dec. 1989.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, Massachusetts, 1993.

[Mil89] R. Milner, *Communication and concurrency*, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[Mol85] M. K. Molloy, "Discrete time stochastic Petri nets," *IEEE Trans. Software Engineering,* 11(4):417–423, April 1985.

[Mur89] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, 77(4):451–480, April 1989.

[Mur89a] T. Murata, B. Shenker, and S. M. Shatz, "Detection of Ada static deadlocks using Petri net invariants," *IEEE Trans. Software Engineering*, 15(3):314–326, March 1989.

[Nic89] D. M. Nicol, "Optimal partitioning of random programs across two processors," *IEEE Trans. Software Engineering*, 15(2):134–141, Feb. 1989.

[Owi82] S. S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," *ACM Trans. Prog. Lang. Syst.*, 4(3):455–495, July 1982.

[Pen97] D. T. Peng, K. G. Shin, and T. F. Abdelzaher, "Assignment and scheduling communicating periodic tasks in distributed real-time systems," *IEEE Trans. Software Engineering*, 23(12):745–758, Dec. 1997.

[Pet81] J. L. Peterson, *Petri net theory and the modeling of systems,* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[PMR99] G.P. Picco, A. L. Murphy, and G.-C. Roman. Lime: Linda meets mobility. In *Proceedings the 21st International Conference on Software Engineering (ICSE-99)*, pages 368–377, Los Angeles, California, May 1999.

[Sha96] S. M. Shatz, S. Tu, T. Murata, and S. Duri. "An Application of Petri Net Reduction for Ada Tasking Deadlock Analysis," *IEEE Trans. Parallel and Distributed Systems*, 7(12):1307–1322, Dec. 1996.

[ST98] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[Sta91] P. H. Starke, "Reachability analysis of Petri nets using symmetries," *Syst. Anal. Model. Simul.*, 8:293–303, 1991.

[Sto77] H. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. Software Engineering*, 3(1):85–93, Jan. 1977.

[Str86] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Trans. Software Engineering,* 12(1):157–171, Jan. 1986.

[Tai91] K. C. Tai, R. H. Carver, and E. E. Obaid, "Debugging concurrent Ada programs by deterministic execution," *IEEE Trans. Software Engineering,* 17(1):45–63, Jan. 1991.

[Tay83] R. N. Taylor, "A general-purpose algorithm for analyzing concurrent programs," *Comm. ACM*, 26(5):362–376, May 1983.

[Tay83a] R. N. Taylor, "Complexity of analyzing the synchronization structure of concurrent programs," *Acta Informatica*, 19:57–84, 1983.

[Tho87] K. S. Thomsen and J. L. Knudsen, "A taxonomy for programming languages with multi-sequential processes," *Journal of Systems and Software*, 7(2):127–140, June 1987.

[Tho97] Tommy Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, 1997.

[Uni83] U. S. Department of Defense, Washington, D. C. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A edition, January 1983.

[Val90] A. Valmari, "A stubborn attack on state explosion," in E. M. Clarke and R .P. Kurshan, editors, *Computer-Aided Verification 90*, Series in Discrete Mathematics and Theoretical Computer Science, vol. 3, American Mathematical Society, Providence, Rhode Island, 1991.

[Wit88] L. Wittie, "Debugging distributed C programs by real time replay," *Proc. Work. Parallel Distributed Debugging,* pp. 57–67, May 1988.

[Yok87] Y. Yokote and M. Tokoro, "Concurrent programming in Concurrent Smalltalk," in A. Yonezawa and M. Tokoro, editors, *Object-oriented concurrent programming,* MIT Press, Cambridge, Massachusetts, pp. 129–158, 1987.