Programming Languages for Distributed Computing Systems

HENRI E. BAL

Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands

JENNIFER G. STEINER

Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands

ANDREW S. TANENBAUM

Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands

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 transactions, 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.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

General Terms: Languages, Design

Additional Key Words and Phrases: Distributed data structures, distributed languages, distributed programming, functional programming, languages for distributed programming, languages for parallel programming, logic programming, object-oriented programming, parallel programming

The research of H. E. Bal was supported in part by the Netherlands Organization for Scientific Research under Grant 125-30-10.

J. G. Steiner's current address: Open Software Foundation, Cambridge, MA.

CONTENTS

INTRODUCTION
1. DISTRIBUTED COMPUTING SYSTEMS
1.1 Classes of Distributed Applications
1.2 Requirements for Distributed Programming Support
1.3 Languages for Distributed Programming
2. LANGUAGE SUPPORT FOR PROGRAMMING DISTRIBUTED SYSTEMS
2.1 Parallelism
2.2 Interprocess Communication and Synchronization
2.3 Partial Failure
3. LANGUAGES FOR PROGRAMMING DISTRIBUTED SYSTEMS
3.1 Languages with Logically Distributed Address Spaces
3.2 Languages with Logically Shared Address Spaces
4. CONCLUSIONS
APPENDIX
ACKNOWLEDGMENTS
REFERENCES

INTRODUCTION

During the past decade, many kinds of distributed computing systems have been proposed and built. These systems cover a wide spectrum in terms of design goals, size, performance, and applications. They also differ considerably in how they are programmed. Some are programmed in conventional languages, possibly supplemented with a few new library routines. Others are programmed in completely new languages, specially designed for distributed applications. It is the intention of this paper to describe and compare the methods and languages that can be used for programming distributed systems, and to present in some detail several languages representative of the research to date in this area.

There is no consensus in the literature as to the definition of a distributed computing system, so we begin Section 1 by defining our use of this term. We then discuss the different kinds of distributed computing systems that have been built, the types of applications for which they are intended, and the programming support required for implementing these applications. This programming support may be provided either by the operating system or by a programming language. We briefly examine the first option, and note several disadvantages.

In Section 2 we discuss the second option—special language support for programming distributed computing systems. We identify three issues that must be addressed by a language intended to support distributed applications programming: the ability to execute different pieces of a program on different processors, the ability for these pieces to cooperate with one another, and the ability to cope with (or take advantage of) partial failure of the distributed system. We show how different languages have addressed these issues in very different ways, and how the appropriateness of one language over another depends primarily on the type of application to be written and, to a lesser extent, on the kind of distributed system on which the application is to be implemented.

In Section 3 we look at some representative programming languages designed for distributed computing systems. We divide the languages into simple categories, describing one or two examples from each category in some detail. We hope in this way to give a flavor of current research in this area. The languages we describe are CSP, Occam, NIL, Ada,[1] Concurrent C, Distributed Processes, SR, Emerald, Argus, Aeolus, ParAlfl, Concurrent PROLOG, PARLOG, Linda, and Orca. Finally, we present our conclusions and give an extensive bibliography.

1. DISTRIBUTED COMPUTING SYSTEMS

We begin this section with our definition of a distributed computing system. Noting that there is a spectrum of such systems, characterized by their interconnecting network. We then discuss the kinds of applications for which these systems are used. Next, we list the requirements for supporting the programming of these applications.

[1] Ada is a registered trademark of the U.S. Department of Defense.

Finally, we discuss how programming languages can fulfill these requirements.

There is considerable disagreement in the literature as to what constitutes a distributed system. Among the many definitions that we have seen, there is only one point of agreement: They all require the presence of multiple processors. The confusion may therefore be due to the large number of different architectural models one finds in multiple-processor systems. *Vector computers*, for example, use many processors that simultaneously apply the same arithmetic operations to different data [Russell 1978]. They are best suited for computation-intensive numerical applications. *Dataflow* and *reduction machines* apply different operations to different data [Treleaven et al. 1982]. *Multiprocessors* consist of several autonomous processors sharing a common primary memory [Jones and Schwarz 1980]. These are well suited for running different subtasks of the same program simultaneously. *Multicomputers* are similar to multiprocessors, except that the processors do not share memory, but rather communicate by sending messages over a communications network [Athas and Seitz 1988]. As a final example, there are systems comprised of workstations or minicomputers connected by a local- or wide-area network. This type of system is frequently the target for distributed operating systems [Tanenbaum and van Renesse 1985]. (We will refer to these latter two systems as *workstation-LANs* and *workstation-WANs*.)

Experts strongly disagree as to which of these multiple processor architectures are to be considered distributed systems. Some people claim that all of the configurations mentioned above fall under this category. Others include only geographically distributed computers connected by a wide-area network. Each combination in between these two extremes probably also has its defenders. The meaning we intend to convey by our use of the term *distributed computing system* is the following:

Definition. A distributed computing system consists of multiple autonomous processors that do not share primary memory, but cooperate by sending messages over a communications network.

Each processor in such a system executes its own instruction stream(s) and uses its own local data, both stored in its local memory. Occasionally, processors may need to exchange data; they do so by sending messages to each other over a network. Many different types of networks exist (e.g., hypercube, local-area network, wide-area networks), as will be discussed below. Although these networks have very different physical properties, they all fit into the same model: each is a medium for transferring messages among processors (see Figure 1a). Distributed systems can be contrasted with multiprocessors, in which processors communicate through a shared memory (see Figure 1b).

Of the architectures mentioned in the list of examples above, multicomputers, workstation-LANs, and workstation-WANs qualify as distributed computing systems by our definition.

Distributed systems can be further characterized by their communications networks. The network determines the speed and reliability of interprocessor communication, and the spatial distribution of the processors. Traditionally, a distributed architecture in which communication is fast and reliable and where processors are physically close to one another is said to be *closely coupled*; systems with slow and unreliable communication between processors that are physically dispersed are termed *loosely coupled.*[2]

Closely coupled distributed systems use a communications network consisting of fast, reliable point-to-point links, which connect each processor to some subset of the other processors. Examples of such systems are the Cosmic Cube [Seitz 1985], hypercubes [Ranka et al. 1988], and transputer networks [May and Shepherd 1984]. Communication costs for this type of system used to be on the order of a millisecond, but are expected to drop to less than a

[2] It must be noted that it is not entirely correct to associate these two attributes with *architectures*, as, for example, communication speed also depends very much on the current hardware technology.

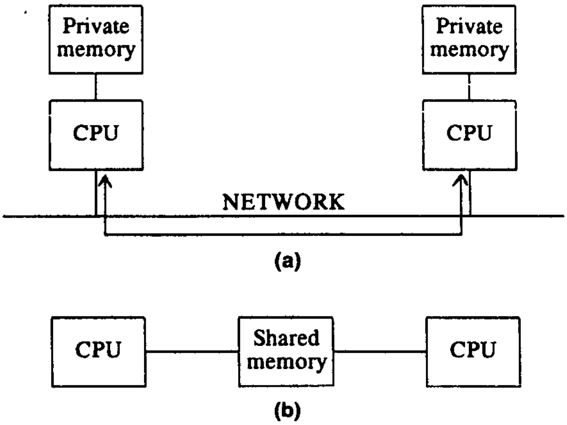

Figure 1. Communication in distributed systems (a) versus shared-memory multiprocessors (b): (a) physical communication by message passing; (b) physical communication through shared memory.

microsecond in the near future [Athas and Seitz 1988].

A more loosely coupled type of distributed system is a workstation-LAN. The local-area network (LAN) allows direct communication between any two processors. Communication cost is typically on the order of milliseconds. In many LANs, communication is not totally reliable. Occasionally, a message may be damaged, arrive out of order, or not arrive at its destination at all. Software protocols must be used to implement reliable communication.

A LAN limits the physical distance between processors to on the order of a few kilometers. To interconnect processors that are farther apart, a wide-area network (WAN) can be used. A workstation-WAN can be seen as a very loosely coupled distributed system. Communication in a WAN is slower and less reliable than in a LAN; communication cost may be on the order of seconds. On the other hand, the increased availability of wide-area lines at speeds above 1 Mbit/s (e.g., T1 lines in the United States), will blur the distinction between LANs and WANs in the future.

In summary, there is a *spectrum* of distributed computing systems, ranging from closely coupled to very loosely coupled systems. Although communication speed and reliability decrease from one end of the spectrum to the other, all systems fit into the same model: autonomous processors connected by some kind of network that communicate by sending messages.

The main purpose of this paper is to study languages for programming the systems that fit into this spectrum. As all systems discussed here are conceptually similar, their programming languages need not be fundamentally different. At least in principle, any one of these languages may be used for programming a variety of distributed architectures. The choice of a suitable language depends very much on the kind of application to be implemented. Below, we first look at several classes of applications that have been written for distributed systems. Then we consider what kind of support is required for these applications and how this support can be provided by a programming language.

1.1 Classes of Distributed Applications

Distributed computing systems are used for many different types of applications. We first look at the reasons why a distributed system might be favored over other architectures, such as uniprocessors or shared-memory multiprocessors, and then classify the distributed applications accordingly. The reasons for programming applications on distributed systems fall into four general categories: decreasing turnaround time for a single computation, increasing reliability and availability, the use of parts of the systems to provide special functionality, and the inherent distribution of the application.

1.1.1 Parallel, High-Performance Applications

Achieving speedup through parallelism is a common reason for running an application on a distributed computing system. By executing different parts of a program on different processors at the same time, some programs will finish faster. In principle, these parallel applications can be run just as well on shared-memory multiprocessors. Shared-memory systems, however, do not scale to large numbers (thousands) of processors, which explains the high interest in implementing parallel programs on distributed systems.

Parallel applications can be further classified by the *grain* of parallelism they use. The grain is the amount of computation time between communications. Large-grain parallel programs spend most of their time doing computations and communicate infrequently; fine-grain parallel programs communicate more frequently.

Fine-grain parallelism and medium-grain parallelism are best applied to closely coupled distributed systems; on loosely coupled systems, the communication overhead becomes prohibitively expensive. The literature contains numerous papers discussing applications that can benefit from this kind of parallelism. Recent introductory papers on this subject are Athas and Seitz [1988] and Ranka et al. [1988].

Large-grain parallelism, on the other hand, is suitable for both closely and loosely coupled distributed systems.[3] Most of the research in this area has focused on implementing large-grain parallel applications on top of existing distributed operating systems [Tanenbaum and van Renesse 1985]. Example applications are compilation of modules of a given program in parallel on different machines [Baalbergen 1988] and implementation of heuristic search algorithms [Bal et al. 1987; Finkel and Manber 1987]. Also, some of the world's best chess programs run on loosely coupled distributed systems. ParaPhoenix, for example, runs on a collection of SUNs connected by an Ethernet [Marsland et al. 1986].

1.1.2 Fault-Tolerant Applications

For critical applications such as controlling an aircraft or an automated factory, a uniprocessor may not be reliable enough. Distributed computing systems are potentially more reliable, because they have the so-called *partial failure* property: since the processors are autonomous, a failure in one processor does not affect the correct functioning of the other processors. Reliability can therefore be increased by replicating the functions or data of the application on several processors. If some of the processors crash, the others can continue the job.

Some fault-tolerant applications may also be run on other multiple-processor architectures that can survive partial failures (e.g., shared-memory multiprocessors). However, if the system must survive natural disasters like fires, earthquakes, and typhoons, one might want the processors to be geographically distributed. To implement a highly reliable banking system, for example, loosely coupled or very loosely coupled distributed systems might be the obvious choice.

Research in this area has focused mainly on software techniques for realizing the potential increase in reliability and availability. Example projects are Circus [Cooper 1985], Clouds [LeBlanc and Wilkes 1985], Argus [Liskov 1988], and Camelot [Spector et al. 1986].

1.1.3 Applications Using Functional Specialization

Some applications are best structured as a collection of specialized services. A distributed operating system like Amoeba, for example, may provide a file service, a print

[3] If the grain of parallelism is large enough, even very loosely coupled distributed systems might be considered for running parallel applications. Recently, an international project was undertaken to find the prime factors of a 100-digit number. The problem was solved in parallel using 400 computers located at research institutes on three different continents (*New York Times*, Oct. 12, 1988).

service, a process service, a terminal service, a time service, a boot service, and a gateway service [Tanenbaum and van Renesse 1985]. It is most natural to implement such an application on distributed hardware. Each service can use one or more dedicated processors, as this will give good performance and high reliability. The services can send requests to each other across the network. If new functions are to be added or if existing functions need extra compute power, it is easy to add new processors. As all processors can communicate through the network, it is easy to share special resources like printers and tape drives.

1.1.4 Inherently Distributed Applications

Finally, there are applications that are inherently distributed. One example is sending electronic mail between people's workstations. The collection of workstations can be regarded as a distributed computing system, so the application (email) has to run on distributed hardware. Similarly, a company with multiple offices and factories may need to set up a distributed system so that people and machines at different sites can communicate.

1.2 Requirements for Distributed Programming Support

We have described a spectrum of distributed architectures and several kinds of applications that may be run on such hardware. We now address the issue of how these applications are to be implemented on these architectures. We refer to this activity as *distributed programming.*

Distributed programming requires support in several areas. We first give the requirements for distributed programming support and then discuss the disadvantages of having these provided by the operating system. Later on, we will see how programming languages designed for distributed systems can fulfill these requirements.

There are basically three issues that distinguish distributed programming from sequential programming:

(1) The use of multiple processors.
(2) The cooperation among the processors.
(3) The potential for partial failure.

Each is discussed below

Distributed programs execute pieces of their code in parallel on different processors. High-performance applications use this parallelism for achieving speedups. Here, the goal is to make optimal use of the available processors; decisions regarding which computations are to run in parallel are of great importance. In fault-tolerant applications, decisions to perform functions on different processors are based on increasing reliability or availability. For special-function and inherently distributed applications, functions may be performed on a given processor because it has certain capabilities or contains needed data. The first requirement for distributed programming support is therefore the ability to assign different parts of a program to be run on different processors.

The processes of a distributed system need to cooperate while executing a distributed application. With parallel applications, processes sometimes have to exchange intermediate results and synchronize their actions. In a system that controls an automated factory, for example, processors have to keep an eye on each other to detect failing processors. The services of a distributed operating system will need each other's assistance: A process service, for example, may need the help of a file service to obtain the binary image file of a process. With distributed electronic mail, messages have to be forwarded between processess. In all these examples, processess must be able to *communicate* and *synchronize* with each other, a second requirement for distributed programming support.

In a uniprocessor system, if the CPU fails, all work ceases instantly. But in a distributed system some CPUs may fail while others continue. This property can be used to write programs that can tolerate hardware failures. This is particularly important for fault-tolerant applications, but it is desirable for other applications as well. For a distributed computer chess program that participates in a tournament, for example, the ability to survive processor failures is highly useful. The third and final requirement for distributed programming support, therefore, is the ability to detect and recover from partial failure of the system.

Ideally, programming support for implementing distributed applications must fulfill all three of these requirements. The support may either be provided by the (distributed) operating system or by a language especially designed for distributed programming. In the first case, applications are programmed in a sequential language extended with library routines that invoke operating-system primitives. As a disadvantage of this approach, the control structures and data types of the sequential language are usually inadequate for distributed programming. Below, we consider two examples of friction between sequential and distributed programming.

Simple actions, like forking off a subprocess or receiving a message from a specific sender, can be expressed relatively easily through library calls. But problems arise, for example, if a process wants selectively to receive a message from one of a number of other processes, where the selection criteria depend on, say, the state of the receiver and the contents of the message. Although concise programming notations exist for such cases (e.g., the select statement discussed in Section 2.2.3), it would probably take a number of complicated library calls to convey such a request to the operating system.

Problems with data types arise if one tries to pass a complex data structure as part of a message to a remote process. As the operating system does not know how data structures are represented, it is unable to pack the data structure into a network packet (i.e., a sequence of bytes). Instead, the programmer has to write explicit code that flattens the data structure into a sequence of bytes on the sending end and that reconstructs the original data structure on the receiving end. A language designed for distributed programming, on the other hand, could do the conversion automatically.

Using a special language for distributed programming also gives other advantages, such as improved readability, portability, and static type checking. Finally and most importantly, a language may present a programming model that is higher level, more. abstract, than the message passing model supported by most operating systems. Several such models are discussed in this paper.

1.3 Languages for Distributed Programming

A central question encountered by developers of distributed software is, "Given a certain application that has to be implemented on a certain distributed computing system, which programming language should be used?" A language can be considered as a candidate if

(1) the language is suitable for the application, and
(2) the language can be implemented with reasonable efficiency on the given hardware.

A maze of languages for distributed programming has evolved during the past decade, making the choice of the most suitable language a difficult one. Most importantly, the underlying *models* of the languages differ widely. Below, we look at several such models. We begin by describing the basic model, which is characterized by the use of processes, message passing, and explicit failure detection. Next, we look at alternative ways for dealing with parallelism, communication, and processor failures.

The most basic model is that of a group of sequential processes running in parallel and communicating through message passing. This model directly reflects the distributed architecture, consisting of processors connected through a communications network. Languages based on this model include CSP and Occam.[4] The language may ease the programming task in many ways,

[4] All languages mentioned in this section are described in Section 3.

for example, by supporting different kinds of message passing (as discussed in Section 2), by masking communication errors, and by type checking the contents of messages. Such languages usually provide a simple mechanism for detecting failures in processors (e.g., an exception is generated or an error returned on attempt to communicate with a faulty processor). An example of a language supporting such features is SR.

For many applications, this basic model of processes and message passing may be just what is needed. The model can be mapped efficiently onto the distributed architecture, and it gives the programmer full control over the hardware resources (processors and network). For other applications, however, the basic model may be too low level. Therefore, several alternative models have been designed for parallelism, communication, and partial failures, which provide higher level abstractions. Below, we give some examples of other models.

Several researchers have come to believe that imperative (algorithmic) languages are not the best ones for dealing with parallelism. Because of the "one-word-at-a-time" von Neumann bottleneck [Backus 1978], imperative languages are claimed to be inherently sequential. This has led to research on parallelism in languages with inherent parallelism, like functional, logic, and object-oriented languages. The lack of side effects in functional languages (like ParAlfl) allows expressions to be evaluated in any order, including in parallel. In logic languages, different parts of a proof procedure can be worked on in parallel, as exemplified by Concurrent PROLOG and PARLOG. Parallelism can also be introduced into object-oriented (or object-based) languages, by making objects active; this approach is taken in Emerald. As a result, models for expressing parallelism that are quite different from the basic model have been developed. The parallelism in these models is usually much more fine grain than in the basic model, however. These languages can be made suitable for large-grain distributed architectures by supplementing them with mapping notations, as discussed in Section 2.1.2.

Likewise, some people are dissatisfied with message passing as the basic communication primitive and have developed communication models that do not directly reflect the hardware communication model. One step in this direction is to have processors communicate through a (generalized form of) procedure call [Birrell and Nelson 1984]; this approach is used in Distributed Processes. A more fundamental break with message passing is achieved through communication models based on shared data. Although implemented on a physically distributed system, such shared data systems are logically nondistributed.

Let us make the following distinction between *logical* and *physical* distribution: As discussed above, distributed computing systems do not have shared memory; the hardware of such systems is *physically distributed.* Distributed systems can be contrasted with multiprocessors or uniprocessors, which have a single systemwide primary memory; these systems are *physically nondistributed.*

A similar distinction can be used for classifying software systems, only here the distinction concerns the logical distribution of the data used by the software, rather than the physical distribution of the memories. For software systems the distinction is *logical* rather than physical. We define a logically distributed system as follows:

Definition. A logically distributed software system consists of multiple software processes that communicate by explicit message passing.

This is in contrast with a logically nondistributed software system, in which software processes communicate through shared data.

There are four different combinations of logical and physical distribution, each of which is viable:

(1) logically distributed software running on physically distributed hardware,
(2) logically distributed software running on physically nondistributed hardware,
(3) logically nondistributed software running on physically distributed hardware, and
(4) logically nondistributed software running on physically nondistributed hardware.

Let us briefly examine each of these. The first class is simple. A typical example is a collection of processes, each running on a separate processor and communicating using SEND and RECEIVE primitives that send messages over a network (e.g., a Hypercube network, LAN, or WAN). The second class has the same logical multiple-process structure, only now the physical message passing is simulated by implementing message passing using shared memory. The third class tries to hide the physical distribution by making the system look like it has shared memory with the programmer. Finally, the fourth class also uses communication through shared data, only the existence of physical shared memory makes the implementation much easier.

In this paper we discuss languages for physically distributed systems. Most of these languages are based on logical distribution. Several others, however, are logically nondistributed and allow processes to communicate through some form of shared data [Bal and Tanenbaum 1988]. In such languages, the implementation rather than the programmer deals with the physical distribution of data over several processors. One example in this class is Linda, which supports an abstract global memory called the Tuple Space. Another example is Orca, which allows processes to share variables of abstract data types (objects). Other members of this class are parallel logic languages (e.g., Concurrent PROLOG and PARLOG) and parallel functional languages (e.g., ParAlfl).

The third important issue in the design of a model for distributed programming—besides parallelism and communication—is handling of processor failures. The basic method for dealing with such failures is to provide a mechanism for failure detection. With this approach, the programmer is responsible for cleaning up the mess that results after a processor crash. The major problem is to bring the system back into a consistent state. This usually can only be done if processor crashes are anticipated and precautions are taken during normal computations (e.g., each process may have to dump its internal state on secondary storage at regular intervals). To release the programmer from all these details, models have been suggested to make recovery from failures easier. Ideally, the system should hide all processor failures from the programmer. Such models have in fact been implemented [Borg et al. 1983]. Alternatively, the programmer can be given high-level mechanisms for expressing which processes and data are important and how they should be recovered after a crash. Languages that use this approach are Argus and Aeolus.

Which model of parallelism, interprocess cooperation, and fault tolerance is most appropriate for a certain application depends very much on the application itself. A distributed system that controls an aircraft can probably do very well without fancy constructs for parallelism. In a distributed banking system, the programmer may want to "see" the distribution of the hardware, so a language that hides this distribution would be most inappropriate. Finally, it makes no sense to apply expensive techniques for fault tolerance to a parallel matrix-multiplication batch-program that takes only a few seconds to execute. On the other hand, there also are numerous cases where these models are useful.

In the next section, we survey current research in language models and notations for distributed programming. Although we discuss a variety of language primitives, one should keep in mind that all primitives are different solutions to the same three problems: dealing with parallelism, communication, and partial failures in distributed computing systems.

2. LANGUAGE SUPPORT FOR PROGRAMMING DISTRIBUTED SYSTEMS

In the previous section, we discussed our definition of the term *distributed computing system* and described the kinds of tasks that might profitably be applied to these systems. We outlined the support required for programming such applications, and what kinds of languages might be expected to provide it. Before describing several of these languages in detail in Section 3, we take this section to discuss in a general way the methods that can be used by

Table 1. Overview of Language-Primitives Discussed in Section 2

Primitive	Example languages
PARALLELISM	
Expressing parallelism	
Processes	Ada, Concurrent C, Linda, NIL
Objects	Emerald, ConcurrentSmalltalk
Statements	Occam
Expressions	ParAlfl, FX-87
Clauses	Concurrent PROLOG, PARLOG
Mapping	
Static	Occam, StarMod
Dynamic	Concurrent PROLOG, ParAlfl
Migration	Emerald
COMMUNICATION	
Message passing	
Point-to-point messages	CSP, Occam, NIL
Rendezvous	Ada, Concurrent C
Remote procedure call	DP, Concurrent CLU, LYNX
One-to-many messages	BSP, StarMod
Data sharing	
Distributed data structures	Linda, Orca
Shared logical variables	Concurrent PROLOG, PARLOG
Nondeterminism	
Select statement	CSP, Occam, Ada, Concurrent C, SR
Guarded Horn clauses	Concurrent PROLOG, PARLOG
PARTIAL FAILURES	
Failure detection	Ada, SR
Atomic transactions	Argus, Aeolus, Avalon
Transparent fault tolerance	NIL

programming languages to fulfill the requirements set out in the preceding section.

As mentioned above, there are three issues that must be addressed in designing a language for distributed programming, above and beyond other programming language issues. These are parallel execution, communication and synchronization between parallel parts of the program, and exceptional conditions brought about by partial failure of the system. As we shall see, each of these issues may be addressed to a greater or lesser degree in a given language, and may be resolved in quite different ways, often depending on the class of distributed application for which the language is intended. Table 1 gives an overview of the primitives described in this section, together with some examples of languages that use the primitives.

2.1 Parallelism

The first issue that must be dealt with in a language for distributed programming is parallel execution. Since a distributed system has by definition more than one processor, it is possible to have more than one part of a program running at the same time. This is what we mean by parallelism.

We begin by drawing a distinction between true parallelism and what we call *pseudoparallelism.* It is sometimes useful to express a program as a collection of processes running in parallel, whether or not these processes actually run at the same time on different processors. For example, a given problem might lend itself well to being expressed as several largely independent processes, running logically in parallel, even though the program may in fact be run on a uniprocessor with only one piece of it running at a given moment in time. The MINIX operating system, for example, was built using this approach [Tanenbaum 1987]. We call this *pseudoparallelism.*[5] This technique has been

[5] Some authors (e.g., Scott [1985]) use the term *concurrency* for denoting pseudoparallel execution. Other authors use the term as a synonym for (real) parallelism, however, so we will not use this term.

employed in programming languages, especially those intended for writing uniprocessor operating systems, for quite some time.

Pseudoparallelism is just as useful in distributed programming as it is in uniprocessor programming. But the difference between true parallelism and pseudoparallelism must be kept in mind, despite the fact that in some languages the distinction is hidden from the programmer. For example, if a program consists of four processes and is running on a distributed system of four or more available processors, the four processes may run in truly parallel fashion—one on each processor. On the other hand, the same program may be running on a system with only two processors, in which case two processes may be assigned to run on each of the two processors. In this case, there are two processes running in pseudoparallel on each processor. At a given point in time, at most two of the program's four processes are running truly in parallel.

In some languages, the distinction between parallelism and pseudoparallelism is not hidden from the programmer. It may be possible for the programmer to explicitly assign (or *map*) pieces of programs to processing units. This delivers more complexity into the hands of the programmer, but also provides more flexibility. For example, given a language in which the programmer controls the mapping of processes onto processors, it is possible to support shared variables among processes known to be running on the same processor, and to disallow the sharing of variables between processes assigned to different processors. This is the case with several languages discussed in Section 3 (e.g., SR, Argus).

The granularity of parallelism varies from language to language, as mentioned above. The *unit of parallelism* in a language ranges from the process (e.g., in Concurrent C) to the expression (in ParAlfl and others). In general, the higher the cost of communication in a distributed system, the larger the appropriate granularity of parallelism. For example, it may be possible to efficiently support fine-grained parallelism in a distributed system with low communication costs, such as a hypercube; whereas in a system with high communication costs, such as a WAN, the communication cost of fine-grained parallelism may outweigh the gain in parallel computation.

Note that the *fact* of parallelism is distinct from parallelism as an *objective*. That is, in some applications, a high degree of parallelism is a goal, as it results in shortened computing time for an application. However, not all distributed applications have high parallelism as their main objective. Yet, even in these latter applications, the ability to express parallelism may be important, since this reflects what is actually occurring in the distributed system.

Finally, not all languages support explicit control of parallelism. In some languages, the dividing up of code into parallel segments is done by the compiler rather than the programmer. Moreover, in some languages the sending of a message on behalf of one process results in the implicit generation of another, parallel process on the remote host to handle the request.

Below we describe several ways in which parallelism can be expressed in programming languages for distributed systems. We then discuss the mapping of parallel computations to physical processors. For a discussion of the expression of pseudoparallelism, we refer the reader to Andrews and Schneider [1983].

2.1.1 Expressing Parallelism

Parallelism can be expressed in a variety of ways. An important factor is the language's *unit of parallelism*. In a sequential language, the unit of parallelism is the whole program. In a language for distributed programming, however, the unit of parallelism can be a process, an object, a statement, an expression, or a clause (in logic languages). We discuss each of these in turn, beginning with the process, as it is intuitively the most obvious.

Processes. In most procedural languages for distributed programming, parallelism is based on the notion of a *process*. Different languages have different definitions of this notion, but in general a process is a logical processor that executes code sequentially and has its own state and data. Processes (or process types) are declared, just like procedures (and procedure types).

Processes are created either implicitly by their declaration or explicitly by some **create** construct. With implicit creation, one usually first declares a process type and then creates processes by declaring variables of that type. Often, *arrays* of processes may be declared. In some languages based on implicit process creation, the total number of processes is fixed at compile time. This makes the efficient mapping of processes onto physical processors easier, but it imposes a restriction on the kinds of applications that can be implemented in the language, since it requires that the number of processes be known in advance.

Having an explicit construct for creating processes allows more flexibility than implicit process creation. For example, the creation construct may allow parameters to be passed to the newly created process. These are typically used for setting up communication channels between processes. If processes do not take parameters (as in Ada [U.S. Department of Defense 1983]), the parameters have to be passed to the newly created process using explicit communication. A mechanism is needed to set up the communication channel over which the parameters are sent.

Another important issue is *termination* of processes. Processes usually terminate themselves, but some primitive may be provided to abort other processes too. Some precautions may be needed to prevent processes from trying to communicate with a terminated process. In Section 2.2.3 we discuss mechanisms for cooperative termination of multiple processes.

Objects. The notion *object-oriented programming* causes as much confusion as the term *distributed system.* In general, an object is a self-contained unit that encapsulates both *data* and *behavior*, and that interacts with the outside world (i.e., other objects) exclusively through some form of message passing. The data contained in the object are visible only within the object itself. The behavior of an object is defined by its *class*, which comprises a list of operations that can be invoked by sending a message to the object. *Inheritance* allows a class to be defined as an extension of another (previously defined) class. Languages that support objects but lack inheritance are usually said to be *object based.*

Objects are primarily intended for structuring programs in a clean and understandable way, reflecting the structure of the problem to be solved as much as possible. At least two different opinions exist on what should be treated as an object. The Smalltalk-80[6] view is simply to consider everything an object, even integers and Booleans [Goldberg and Robson 1983]. The second view (e.g., taken in Aeolus [Wilkes and LeBlanc 1986]) is less pure and lets programmers decide what objects are.

Parallelism in object-oriented languages can be obtained in one of two ways. Smalltalk-80 includes the traditional notion of a process and lets the programmer deal with two kinds of modules: objects and processes. A more orthogonal approach is to use the object itself as the unit of parallelism.

Sequential object-oriented languages are based on a model of *passive* objects. An object is activated when it receives a message from another object. While the receiver of the message is active, the sender is waiting for the result, so the sender is passive. After returning the result, the receiver becomes passive again and the sender continues. At any point of time, only one object in the system is active. Parallelism can be obtained by extending the sequential object model in any of the following ways:

(1) Allow an object to be active without having received a message,
(2) allow the receiving object to continue execution after it returns its result,
(3) send messages to several objects at once, or
(4) allow the sender of a message to proceed in parallel with the receiver.

[6] Smalltalk-80 is a trademark of ParcPlace Systems.

Methods (1) and (2) effectively assign a parallel process to each object, resulting in a model based on *active* objects. Method (4) can be implemented using asynchronous message passing (instead of synchronous message passing) or by letting a single object consist of multiple threads of control.

Parallel Statements. Another way of expressing parallelism is by grouping together statements that are to be executed in parallel. Occam [Inmos Ltd. 1984] allows consecutive statements to be executed either sequentially, as in

```
SEQ
  S1
  S2
```

or in parallel, as in the following:

```
PAR
  S1
  S2
```

This method is easy to use and understand. Initiation and termination of parallel computations are well defined. However, this method gives little support for the structuring of large parallel programs.

The parallel statement described above creates only a fixed number of parallel units. Another method is to use a parallel *loop* statement. Occam contains a parallel **for** statement, similar to a traditional **for** statement, except that all iterations of the loop are executed in parallel, as in the following:

```
PAR i = 0 FOR n
  A[i] := A[i] + 1
```

Although this construct is easy to use, it is not as general as other mechanisms.

Functional Parallelism. In a pure functional (applicative) language, functions behave as mathematical functions: They compute a result that depends only on the values of their input data. Such functions do not have any side effects. In contrast, procedural (imperative) languages allow functions to affect each other in various ways, for example, through global variables or pointer variables. Procedural languages are claimed to be more flexible, whereas functional languages have a sounder mathematical basis. We will not enter into the holy war between these two schools of thought, but we will concentrate on the way functional languages can be used for programming distributed systems.

If functions do not have any side effects, it makes no difference (except perhaps for termination) in which order they are executed. For example, in the expression

$h(f(3, 4), g(8))$

it is irrelevant whether f or g is evaluated first. Consequently, it is possible to evaluate f and g in parallel. In principle, all function calls can be executed in parallel, the only restriction being that a function using the result of another function wait for that result to become available (e.g., h waits for f and g). This implicit parallelism is fine grained and is well suited for achitectures supporting such parallelism, such as data-flow computers. Several data-flow languages are based on this principle, for example, Id and VAL [Ackerman 1982].

For distributed systems (and to some extent also for other architectures), the functional approach has some problems that need to be resolved. First of all, blindly evaluating all functions in parallel is not a very good idea. If a function does relatively little work (such as adding two integers), the overhead of doing it in parallel and communicating the result back to the caller will far outweigh the savings in elapsed computation time. If a certain function call is selected for remote execution, there still remains the choice between evaluating its arguments either locally (and then sending them to the remote processor) or remotely (by dispatching the unevaluated expressions).

Ideally the compiler should analyze the program and decide on which processor to perform each function call. Since current compilers are not yet capable of taking maximum advantage of parallelism in this way, mechanisms have been proposed to put control in the hands of the programmer [Burton 1984; Hudak 1986].

AND/OR-Parallelism. Logic programming offers many opportunities for parallelism [Takeuchi and Furukawa 1986]. We describe AND/OR parallelism, as this mechanism is suitable for distributed programming and has been incorporated into many parallel logic programming languages.

Logic programs can be read *declaratively* as well as *procedurally*. In the code below, two *clauses* for the predicate A are given:

(1) $A :- B, C, D.$
(2) $A :- E, F.$

The declarative reading of the clauses is "if B, C, and D are true, then A is true" (Clause (1)) and "if E and F are true, then A is true" (Clause (2)). Procedurally, the clauses can be interpreted as "to prove theorem A, you either have to prove subtheorems (or *goals*) B, C, and D, or you have to prove subtheorems E and F." From the procedural reading, it becomes clear that there are two opportunities for parallelism:

(1) The two clauses for A can be worked on in parallel, until one of them succeeds, or both fail.
(2) For each of the two clauses, the subtheorems can be worked on in parallel, until they all succeed, or any one of them fails.

The former kind of parallelism is called *OR-parallelism*; the latter is called *AND-parallelism.*

The parallel execution of a logic program can also be described in terms of processes, resulting in a third interpretation, the *process reading*, of logic programs. If we associate a separate process with every subtheorem to be proved, then Clause (1) simply states that a process trying to prove A can be replaced by three parallel processes that try to prove B, C, and D. In general, a clause like

$P_0 :- P_1, \ldots, P_N$

causes a single process to be replaced by N other processes. If $N = 0$, the original process *terminates*. For $N = 1$, the process effectively changes its state, going to work on a different goal. If $N > 1$, then $(N - 1)$ new processes are created. Such processes are very lightweight and similar in granularity to a procedure call in a procedural language.

If the goals of a clause share some variables, they cannot be evaluated independently, because *conflicts* may arise when several goals try to generate a value for a shared variable. For example, in the clause

$A :- B(X), C(X)$

the variable X creates a dependency between the goals B and C. Several approaches have been suggested to deal with this problem. One method is to have the programmer restrict the rights of goals to instantiate (or *bind*) shared variables. In Concurrent PROLOG [Shapiro 1986], the notation

$A :- B(X), C(X?)$

indicates that B is allowed to generate a binding for X, but C is only allowed to read X. This mechanism can be used for interprocess communication and synchronization, as discussed in Section 2.2.2. Another method for dealing with conflicts is to solve dependent goals sequentially. In general, both compile-time analysis and run-time checks are used to determine if two clauses are independent. Both solutions—restricted instantiation and sequential solution of dependent goals—necessarily restrict parallelism.

2.1.2 Mapping Parallel Computations onto Physical Processors

In the previous section, we described several ways in which languages for distributed programming can provide support for the expression of parallelism. A related issue is how these parallel computations are distributed over the available physical processors, in other words, which parallel unit is executed on which processor at a given point in time. We refer to the assignment of computations to processors as *mapping.* Some languages give the programmer control over mapping, and in this section we describe some ways in which this can be expressed.

Mapping strategies vary depending on the application to be implemented. The assignment of processes to processors will be quite different in an application whose objective is to obtain maximum speedup through parallelism, and in an application whose objective is to obtain high availability through replication, for example.

When the goal of a distributed program is to speed up computation time through parallelism, the mapping of processes to processors is similar to load balancing in distributed operating systems: Both attempt to maximize parallelism through efficient use of available computing power. But there are important differences. An operating system tries to distribute the available processing power *fairly* among competing processes from different programs and different users. It may try to reduce communication costs by having processes that communicate frequently run in pseudoparallel on the same processor. The goal of mapping, however, is to minimize the execution time of a single distributed program. As all parallel units are part of the same program, they are cooperating rather than competing, so fairness need not be an issue. In addition, the reduction of communication overhead achieved through mapping processes to the same processor must be weighed against the resulting loss of parallelism [Kruatrachue and Lewis 1988].

If the application's goal is to increase fault tolerance, an entirely different mapping strategy may be taken. Processes may be replicated to increase availability. The mapping strategy should at least assign the replicas of the same logical process to different physical processors.

An important choice in the design of a parallel language is whether mapping will be under user control. If not, mapping is done transparently by the compiler and language run-time system, possibly assisted by the operating system. At first sight, this may ease the programmer's task, but the system generally does not have any knowledge about the problem being implemented, so problem-specific mapping strategies would be ruled out. This is a severe restriction for many applications.

Programmable (i.e., user-controlled) mappings usually consist of two steps. In the first step, the parallel units are mapped onto the physical processors. Several parallel units may be mapped onto the same processor. In the second step, the units on the same processor are scheduled by a local mapping, usually based on *priorities* assigned to the parallel units.

There are three approaches for assigning parallel units to processors, whether the assignment is done by the programmer or the system: The processor can either be fixed at compile time, fixed at run time, or not fixed at all. The first method is least flexible, but has the distinct advantage that it is known at compile time which parallel units will run on the same processor, allowing the programmer to take advantage of the fact that these processes will have shared memory available. StarMod uses the notion of a *processor module* that groups together processes located on the same processor [Cook 1980]. These processes are allowed to communicate through shared variables, whereas communication between processes on different processors is restricted to message passing.

With the run-time approach to mapping computations to processors, a parallel unit is assigned to a processor when that unit is created. An example is the Turtle notation designed by Shapiro for executing Concurrent PROLOG programs on an infinite grid of processors, where each processor can communicate with its four neighbors [Shapiro 1984]. Every process has a *position* and a *heading*, just like a Turtle in the LOGO programming language [Papert 1981]. By default, the position and heading of a process are those of its parent (creator), but they can be altered using a sequence of Turtle commands. For example, if a process located on Processor P and heading northward uses the rule

A :− B, C @ (left, forward),
D @ (right, forward).

to solve A, then Process B is created on Processor P, Process C is created on the processor to the west of P, and Process D is created on the processor to the east of P.

B is headed northward, *C* westward, and *D* eastward.[7]

Only a few languages support the third approach to processor allocation, allowing a process to execute on different processors during its lifetime. Emerald, for example, is an object-based language that allows objects to migrate from one processor to another [Jul et al. 1988]. The language has primitives to determine the current location of an object, to fix or unfix an object on a specific processor, and to move an object to a different processor.

2.2 Interprocess Communication and Synchronization

The second issue that must be addressed in the design of a language for distributed programming is how the pieces of a program that are running in parallel on different processors are going to cooperate. This cooperation involves two types of interaction: communication and synchronization. For example, Process *A* may require some data *X* that is the result of some computation performed by Process *B*. There must be some way of getting *X* from *B* to *A*. In addition, if Process *A* comes to the point in its execution that requires the information *X* from Process *B*, but Process *B* has not yet communicated the information to *A* for whatever reason, *A* must be able to wait for it. Synchronization and communication mechanisms are closely related, and we treat them together.

An issue related to synchronization is *nondeterminism*. A process may want to wait for information from any of a group of other processes, rather than from one specific process. As it is not known in advance which member (or members) of the group will have its information available first, such behavior is nondeterministic. In some cases it is useful to dynamically control the group of processes from which to take input. For example, a buffer process may accept a request from a producer process to store an item in the buffer whenever the buffer is not full; it may accept a request from a consumer process to add an item whenever the buffer is not empty. To program such behavior, a notation is needed to express and control nondeterminism. We look at such notations in Section 2.2.3.

Expression of interprocess communication (IPC)[8] falls into two general categories—shared data and message passing—although this categorization is not always clear-cut. Parallel logic languages that provide shared logical variables, for example, are frequently used for programming in a message-passing style (see Section 2.2.2). Note that the *model* provided by the language for expressing IPC and the *implementation* of that model may be two entirely different things; in particular, since we restrict our discussion to languages for systems without shared memory, any shared data model must be simulated in the language implementation.

2.2.1 Message Passing

We first discuss communication through message passing. Many factors come into play in the sending of a message: who sends it, what is sent, to whom is it sent, is it guaranteed to have arrived at the remote host, is it guaranteed to have been accepted by the remote process, is there a reply (or several replies), and what happens if something goes wrong. There are also many considerations involved in the receipt of a message: for which process or processes on the host, if any, is the message intended; is a process to be created to handle this message; if the message is intended for an existing process, what happens if the process is busy—is the message queued or discarded; and if a receiving process has more than one outstanding message waiting to be serviced, can it choose the order in which it services messages—be it FIFO, by sender, by some message type or identifier, by the contents of the message, or according to the receiving process's internal state.

[7] This Turtle notation was later generalized into a layered method, using virtual machines [Taylor et al. 1987a]. The layered method is also suitable for other architectures than a processor grid.

[8] We adopt the well-known term *interprocess communication* although it is somewhat misleading, since the unit of parallelism is not always the process, as has been discussed above. In the rest of this section, we will use the term *process* as a shorthand for *unit of parallelism*.

We begin with a general discussion of issues common to all message-passing mechanisms. We then outline four specific message-passing models: point-to-point messages, rendezvous, remote procedure call, and one-to-many messages.

General Issues. The most elementary primitive for message-based interaction is the point-to-point message from one process (the sender) to another process (the receiver). Languages usually provide only *reliable* message passing. The language run-time system (or the underlying operating system) automatically generates acknowledgment messages, transparent at the language level.

Most (but not all) message-based interactions involve two parties, one sender and one receiver. The sender initiates the interaction *explicitly*, for example, by sending a message or invoking a remote procedure. On the other hand, the receipt of a message may either be *explicit* or *implicit.* With explicit receipt, the receiver is executing some sort of *accept* statement specifying which messages to accept and what actions to undertake when a message arrives. With implicit receipt, code is automatically invoked within the receiver. It usually creates a new thread of control within the receiving process. Whether the message is received implicitly or explicitly is transparent to the sender.

Explicit message receipt gives the receiver more control over the acceptance of messages. The receiver can be in many different *states* and accept different types of messages in each state. More accurate control is possible if the accept statement allows messages to be accepted conditionally, depending on the arguments of the message (as in SR [Andrews 1981] and Concurrent C [Gehani and Roome 1989]). A file server, for example, may want to accept a request to open a file only if the file is not locked. In Concurrent C this can be coded as follows:

```
accept open(f) suchthat not_locked(f) {
   . . .
   process open request
   . . .
}
```

Some languages give the programmer control over the *order* of message acceptance. Usually, messages are accepted in FIFO order, but occasionally it is useful to change this order according to the type, sender, or contents of a message. For example, the file server may want to handle read requests for small amounts of data first:

```
accept read(f, offset, nr_bytes)
by nr_bytes {
   . . .
   process read request
   . . .
}
```

The value given in the **by** expression determines the order of acceptance. If conditional or ordered acceptance is not supported by the language, an application needing these features will have to keep track of requests that have been accepted but not handled yet.

Another major issue in message passing is *naming* (or addressing) of the parties involved in an interaction: to whom does the sender wish to send its message and, conversely, from whom does the receiver wish to accept a message? These parties can be named *directly* or *indirectly.* Direct naming is used to denote one specific process. The name can be the static name of the process or an expression evaluated at run time. A communication scheme based on direct naming is *symmetric* if both the sender and the receiver name each other. In an *asymmetric* scheme, only the sender names the receiver. In this case, the receiver is willing to interact with any sender. Note that interactions using implicit receipt of messages are always asymmetric with respect to naming. Direct naming schemes, especially the symmetric ones, leave little room for expressing nondeterministic behavior. Languages using these schemes therefore have a separate mechanism for dealing with nondeterminism (see Section 2.2.3).

Indirect naming involves an intermediate object, usually called a *mailbox*, to which the sender directs its message and to which the receiver listens. In its simplest form, a mailbox is just a global name. More advanced schemes treat mailboxes as values that can be passed around, for example, as part of a message. This option allows highly flexible communication patterns to be expressed. Mailing a letter to a post office box rather than a street address illustrates the difference between indirect and direct naming. A letter sent to a post office box can be collected by anyone who has a key to the box. People can be given access to the box by duplicating keys or by transferring existing keys (possibly through another P.O. box). A street address, on the other hand, does not have this flexibility.

Synchronous and Asynchronous Point-to-Point Messages. The major design issue for a point-to-point message-passing system is the choice between *synchronous* and *asynchronous* message passing. With synchronous message passing, the sender is blocked until the receiver has accepted the message (explicitly or implicitly). Thus, the sender and receiver not only exchange data, but they also synchronize. With asynchronous message passing, the sender does not wait for the receiver to be ready to accept its message. Conceptually, the sender continues immediately after sending the message. The implementation of the language may suspend the sender until the message has at least been copied for transmission, but this delay is not reflected in the semantics.

In the asynchronous model, there are some semantic difficulties to be dealt with. As the sender S does not wait for the receiver R to be ready, there may be several *pending* messages sent by S, but not yet accepted by R. If the message-passing primitive is *order preserving*, R will receive the messages in the order they were sent by S. The pending messages are *buffered* by the language run-time system or the operating system. The problem of a possible buffer overflow can be dealt with in one of two ways. Message transfers can simply fail whenever there is no more buffer space. Unfortunately, this makes message passing less reliable. The second option is to use *flow control*, which means the sender is blocked until the receiver accepts some messages. This introduces a synchronization between the sender and receiver and may result in unexpected deadlocks.

In the synchronous model, there can be only one pending message from any process S to a process R. Usually, no ordering relation is assumed between messages sent by different processes. Buffering problems are less severe in the synchronous model, as a receiver need buffer at most one message from each sender, and additional flow control will not change the semantics of the primitive. On the other hand, the synchronous model also has its disadvantages. Most notably, synchronous message passing is less flexible than asynchronous message passing, because the sender always has to wait for the receiver to accept the message, even if the receiver does not have to return an answer [Gehani 1987].

Rendezvous. A point-to-point message establishes one-way communication between two processes. Many interactions between processes, however, are essentially two-way in nature. For example, in the client/server model the client requests a service from a server and then waits for the result returned by the server. This behavior can be simulated using two point-to-point messages, but a single higher level construct is easier to use and more efficient to implement. We will describe two such constructs, *rendezvous* and *remote procedure call.*

The rendezvous model is based on three concepts: the entry declaration, the entry call, and the accept statement.[9] The entry declaration and accept statement are part of the server code, while the entry call is on the client side. An *entry declaration* syntactically looks like a procedure declaration. An entry has a name and zero or more formal parameters. An *entry call* is similar to a procedure call statement. It names the entry and the process containing the entry, and it supplies actual parameters. An *accept statement* for the entry may contain a list of statements, to be executed when the entry is called, as in the following accept statement for the entry **incr**:

```
accept incr(X: integer; Y: out integer)
  do Y := X + 1;
end;
```

[9] Here we use the terminology introduced by Ada.

An interaction (called a *rendezvous*) between two processes *S* and *R* takes place when *S* calls an entry of *R*, and *R* executes an **accept** statement for that entry. The interaction is fully synchronous, so the first process that is ready to interact waits for the other. When the two processes are synchronized, *R* executes the **do** part of the accept statement. While executing these statements, *R* has access to the input parameters of the entry, supplied by *S*. *R* can assign values to the output parameters, which are passed back to *S*. After *R* has executed the **do** statements, *S* and *R* continue their execution in parallel. *R* may still continue working on the request of *S*, although *S* is no longer blocked.

Remote Procedure Call. Remote procedure call (RPC) is another primitive for two-way communication. It resembles a normal procedure call, except that the caller and receiver are different processes. When a process *S* calls a remote procedure *P* of a process *R*, the input parameters of *P*, supplied by *S*, are sent to *R*. When *R* receives the invocation request, it executes the code of *P* and then passes any output parameters back to *S*. During the execution of *P*, *S* is blocked. *S* is reactivated by the arrival of the output parameters. This is in contrast with the rendezvous mechanism, where the caller is unblocked as soon as the accept statement has been executed. Like rendezvous, RPC is a fully synchronous interaction. Acceptance of a remote call is usually (but not always) *implicit* and creates a new thread of control within the receiver.

A major design choice is between a *transparent* and a *nontransparent* RPC mechanism. Transparent RPC offers semantics close to a normal procedure. This model, advocated by Nelson and Birrell, has significant advantages [Nelson 1981; Birrell and Nelson 1984]. Foremost, it gives the programmer a simple, familiar primitive for interprocess communication and synchronization. It also is a sound basis for porting existing sequential software to distributed systems.

Unfortunately, achieving exactly the same semantics for RPC as for normal procedures is close to impossible [Tanenbaum and van Renesse 1988]. One source of problems is that, in the absence of shared memory, pointers (address values) are meaningless on a remote processor. This makes pointer-valued parameters and call-by-reference parameters highly unattractive. De-referencing a pointer passed by the caller has to be done at the caller's side, which implies extra communication. An alternative implementation is to send a copy of the value pointed at the receiver, but this has subtly different semantics and may be difficult to implement if the pointer points into the middle of a complex data structure, such as a directed graph. In languages lacking strong type checking, it may not even be clear what type of object the pointer points to. Similarly, call-by-reference can be replaced by copy-in/copy-out, but also at the cost of slightly different semantics. The issue of passing arguments to a remote procedure is discussed further by Herlihy and Liskov [1982].

The possibility of processor crashes makes it even more difficult to obtain the same semantics for RPC as for normal procedures. If *S* calls a remote procedure *P* of a process *R* and the processor of *R* crashes before *S* gets the results back, then *S* clearly is in trouble. First, the results *S* is waiting for will never arrive. Second, it is not known whether *R* died before receiving the call, during the execution of *P*, or after executing *P* (but before returning the results). The first problem can be solved using time-outs. The second problem is more serious. If *P* has no side effects, the call can be repeated, perhaps on a different processor or after a certain period of time. If *P* does have side effects (e.g., incrementing a bank account in a database), executing (part of) *P* twice may be undesirable.

Because of these difficulties in achieving normal call semantics for remote calls, Hamilton argues that remote procedures should be treated differently from the start, resulting in a nontransparent RPC mechanism [Hamilton 1984]. Almes describes an RPC implementation in the context of an existing language (Modula-2) and distributed operating system (the V system) [Almes 1986]. Although the goal of the implementation was to make remote calls as similar to normal calls as possible, special features for remote calls had to be added to obtain an efficient implementation. Almes's RPC system therefore is also nontransparent.

One-to-Many Message Passing. Many networks used for distributed computing systems support a fast *broadcast* or *multicast* facility. A broadcast message is sent to *all* processors connected to the network. A multicast message is sent to a specific subset of these processors. It takes about the same time to broadcast or multicast a message as to send it to one specific processor. Unfortunately, it is not guaranteed that messages are actually delivered at all destinations. The hardware attempts to send the messages to all processors involved, but messages may get lost due to communication errors or because some receiving processors were not ready to accept a message.

Despite being unreliable, broadcast and multicast are useful for operating system kernels and language run-time systems. For example, to locate a processor providing a specific service, an enquiry message may be broadcast. In this case, it is not necessary to receive an answer from every host: Just finding one instance of the service is sufficient. Broadcast and multicast are also useful for implementing distributed algorithms, so some languages provide a *one-to-many* message-passing primitive.

One-to-many communication has several advantages over point-to-point message passing. If a process needs to send data to many other processes, a single multicast will be faster than many point-to-point messages. More importantly, a broadcast primitive may guarantee a certain *ordering* of messages that cannot be obtained easily with point-to-point messages [Birman and Joseph 1987]. A broadcast primitive that delivers messages at all destinations in the same order, for example, is highly useful for consistent updating of replicated data [Joseph and Birman 1986; Bal and Tanenbaum 1988]. Finally, broadcasting may lead to new programming styles.

Gehani describes a system of Broadcasting Sequential Processes (BSP) based on CSP and the concept of *broadcast programming* [Gehani 1984b]. In CSP a message is sent to one specific process. In BSP a message can also be sent to *all* processes or to a list of processes. Both primitives are reliable (i.e., messages are delivered at all destinations). If the underlying hardware is not reliable, extra software protocols have to be added by the operating system or language run-time system. Broadcast in BSP is asynchronous, because the sender normally does not want to wait until all other processes are ready to receive a message. Two forms of broadcast are defined. An *unbuffered* broadcast message is only received by those processes ready to accept one. *Buffered* broadcast messages are buffered by the receiving processes, so each process will eventually receive the message. A receiver may accept messages from any process, or it may screen out messages based on their contents or on the identity of the sender (passed as part of the message).

2.2.2 Data Sharing

In the previous section, we discussed models of interprocess communication based on message passing. In this section, we describe how parts of a distributed program can communicate and synchronize through the use of shared data. If two processes have access to the same variable, communication can take place by one process setting the variable and the other process reading it. This is true whether the processes are running on the host where the variable is stored and can manipulate it directly, or if the processes are on different hosts and access the variable by sending a message to the host on which it resides. The use of shared variables for the communication and synchronization of processes running in pseudoparallel on a uniprocessor has been studied extensively. We assume a familiarity with this material; the uninitiated reader is referred to Andrews and Schneider [1983] for an excellent overview.

As mentioned above, many distributed languages support processes running in pseudoparallel on the same processor, and these often use traditional methods of communication and synchronization through shared variables. See, for example, the description of **mutex** in Argus and semaphores in SR in Section 3. What we are interested in here, however, is the use of shared data for communication and synchronization of processes running on different processors.

At first sight it may seem to be unnatural to use shared data for communication in a distributed system, as such systems do not have physically shared memory. However, the shared data paradigm has several advantages (as well as disadvantages) over message passing [Bal and Tanenbaum 1988]. Whereas a message generally transfers information between two specific processes, shared data are accessible by any process. Assignment to shared data conceptually has immediate effect; in contrast, there is a measurable delay between sending a message and its being received. On the other hand, shared data require precautions to prevent multiple processes from simultaneously changing the same data. As neither of the paradigms is universally better than the other one, both paradigms are worth investigating.

Simple shared variables, as used, for example, in Algol 68 [van Wijngaarden et al. 1975], are not well suited for distributed systems. In principle, they can be implemented by simulating shared physical memory, using, for example, a method such as Li's shared virtual memory [Li and Hudak 1986]. None of the languages we know of does this, however, probably due to performance considerations. Several other communication models based on shared data exist, however, that are better suited for distributed systems. These models place certain restrictions on the shared data, making a distributed implementation feasible. Below we describe two methods for providing shared data to distributed processes: distributed data structures and shared logical variables. Both models are used in several languages for distributed programming (see Section 3) that have been implemented on different kinds of distributed architectures. These languages are mainly useful for applications where the programmer need not be aware of the physical distribution of main memory, as discussed in Section 1.

Note that objects, whose role in expressing parallelism was discussed in Section 2.1.1, may also be thought of as implementing shared data in a distributed program. Just as with the shared data models that are discussed in this section, two processes may communicate indirectly with one another by invoking operations on a given object. Objects, since they control access to the data they manage, can also implement synchronization of access to those data by other processes, analogously to the synchronization of pseudoparallel processes accessing data controlled by a monitor.

A different approach to the synchronization of distributed access to shared data is taken by languages that implement atomic transactions. Since this approach also involves dealing with partial failures of the distributed systems, we will treat it in the section on atomic transactions.

Distributed Data Structures. Distributed data structures are data structures that can be manipulated simultaneously by several processes [Carriero et al. 1986]. This paradigm was first introduced in the language Linda, which uses the concept of a *Tuple Space* for implementing distributed data structures [Ahuja et al. 1986]. We will use the Tuple Space model for discussing the distributed data structures paradigm.

The Tuple Space (TS) is conceptually a shared memory, although its implementation does not require physical shared memory. The TS is one global memory shared by all processes of a program [Gelernter 1985]. The elements of TS, called *tuples*, are ordered sequences of values, similar to records in Pascal [Wirth 1971]. For example,

["jones", 31, true]

is a tuple with three fields: a string, an integer, and a Boolean.

Three atomic operations are defined on TS: **out** adds a tuple to TS, **read** reads a tuple contained in TS, and **in** reads a tuple and also deletes it from TS. Unlike normal shared variables, tuples do not have addresses. Rather, tuples are addressed by their contents. A tuple is denoted by specifying either the *value* or the *type* of each field. This is expressed by supplying an *actual* parameter (a value) or a *formal* parameter (a variable) to an operation. For example, if *age* is a variable of type integer and *married* is a variable of type Boolean, then the tuple shown above can be read in the operation

read("jones", **var** age, **var** married)

or read and removed in the operation

in("jones", **var** age, **var** married).

In both operations, the variable *age* is assigned the value of the second field (31) and the variable *married* gets the value of the last field (true). Both the **in** and the **read** operations try to find a matching tuple in TS. A tuple matches if each field has the value or type passed as parameter to the operation. If several matching tuples exist, one is chosen arbitrarily. If there are no matching tuples, the operation (and the invoking process) *blocks* until another process adds a tuple that does match (using **out**).

There is no operation that modifies a tuple in place. To change a tuple, it must first be removed from TS, then modified, and then put back into TS. Each **read**, **in**, or **out** operation is atomic: The effect of several simultaneous operations on the same tuple is the same as that of executing them in some (undefined) sequential order. In particular, if two processes want to remove the same tuple, only one of them will succeed, and the unlucky one will block. These two properties make it possible to build distributed data structures in TS. For example, a distributed array can be built out of tuples of the form [name, index, value]. The value of element *i* of array *A* can be read into a local integer variable *X* with a simple **read** operation:

read("*A*", *i*, **var** *X*)

To assign a new value *Y* to element *i*, the current tuple representing *A*[*i*] is removed first; then a tuple with the new value is generated:

in("*A*", *i*, **var** void)
out("*A*", *i*, *Y*)

To increment element *i*, the current tuple is removed from TS, its value is stored in a temporary variable, and the new value is computed and stored in a new tuple:

in("*A*", *i*, **var** *tmp*)
out("*A*", *i*, *tmp* + 1)

If two processes simultaneously want to increment the same array element, the element will indeed be incremented twice. Only one process will succeed in doing the **in**, and the other process will be blocked until the first one has put the new value of *A*[*i*] back into TS.

In a distributed implementation of TS, the run-time system takes care of the distribution of tuples among the processors. Several strategies are possible, such as replicating the entire TS on all processors, *hashing* tuples onto specific processors, or storing a tuple on the processor that did the **out** operation [Gelernter 1985].

In contrast with interprocess communication accomplished through message passing, communication through distributed data structures is anonymous. A process reading a tuple from TS does not know or care which other process inserted the tuple. Neither did the process executing an **out** on a tuple specify which process the tuple was intended to be read by. This information could in principle be included in a distributed data structure, for example, by having *sender* and *receiver* fields as part of the structure, but it is not an inherent part of the model.

Shared Logical Variables. Another shared data model is the shared logical variable. Logical variables have the "single-assignment" property. Initially, they are unbound, but once they receive a value (by unification), they cannot be changed. In Section 2.1.1 we noted that this property can cause conflicts between parallel processes sharing logical variables. Below, we show how such variables can be used as communication channels between processes.

As an example, assume the three goals of the conjunction

goal_1(X, Y), goal_2(X, Y), goal_3(X)

are solved in parallel by Processes $P1$, $P2$, and $P3$. The variable X (initially unbound) is a communication channel between the three processes. If any of them binds X to a value, the other processes can use this value. Likewise, Y is a channel between $P1$ and $P2$.

Processes synchronize by suspending on unbound variables. If Y is to be used to send a message from $P1$ to $P2$, then $P2$ can suspend until Y is bound by $P1$. There are several ways to realize suspension on shared variables, but the general idea is to restrict the rights of specific processes to generate bindings for variables (i.e., to unify them with anything but an unbound variable). If a process wants to unify two terms, the unification may need to generate a binding for some variables. If the process does not have the right to bind one of these variables, the process suspends until some *other* process that does have this right generates a binding for the variable. The first process can then continue its unification of the two terms. Examples of mechanisms to restrict the rights for binding variables are the *read-only variables* of Concurrent PROLOG [Shapiro 1986] and the *mode declarations* of PARLOG [Clark and Gregory 1986].

At first sight, shared logical variables seem to be capable of transferring only a single message, as bindings cannot be undone. But, in fact, the logical variable allows many communication patterns to be expressed. The key idea is to bind a logical variable to a term containing other (unbound) variables, which can be used as channels for further communication. A logical variable is like a Genie, from which you can ask one wish. What would you ask such a Genie? To have two more wishes! Then use one of them, and iterate.[10]

This idea has been used to develop several programming techniques. For example, a *stream* of messages between a producer and a consumer is created by having the producer bind a shared variable to a list cell with two fields, head and tail. The head is bound to the message, and the tail is the new stream variable, used for subsequent communications (wishes). This is illustrated in Figure 2 where the first call, producer(1, S), will cause S to be bound to [1 | $S1$], where $S1$ is an unbound variable. The next (recursive) call, producer(2, $S1$), binds $S1$ to [4 | $S2$], where $S2$ is unbound. The call, consumer(S), will cause the consumer process to be blocked until S is bound by the producer. When S is bound to [1 | $S1$], the consumer wakes up, calls, use(1), followed by the recursive call, consumer($S1$). The latter call blocks until $S1$ is bound to [4 | $S2$], and so on.

Other techniques implementable with shared logical variables are bounded-buffer streams [Takeuchi and Furukawa 1985], one-to-many streams, and incomplete messages. An incomplete message contains variables that will be bound by the receiver, thus returning reply values. The sender can wait for replies by suspending on such a variable. Incomplete messages can be used to implement many different message protocols (e.g., remote procedures and rendezvous, discussed above) and to dynamically set up communication channels between processes.

The shared logical variable model also has some disadvantages, as discussed by Gelernter [1984]. Only a single process can append to a stream implemented through logical variables (e.g., in Figure 2 only the producer can append to S). Applications based on the client/server model, however, require multiple clients to send messages to a single server (many-to-one communication). To implement this in a parallel logic language, each client must have its own output stream. There are two alternatives for structuring the server: First, the server may use a separate input stream for each client and accept messages sent through each of these streams. This requires the server to know the identities of all clients and thus imposes a limit on the number of clients. The second alternative is to *merge* the output streams of all clients and present it as a single input

[10] This analogy was contributed by Ehud Shapiro.

```
/* the consumer is not allowed to bind S */
mode producer(N?, S^), consumer(S?).

producer(N, [X|Xs]) :-   /* produce stream of squares */
   X is N*N, N2 is N+1, producer(N2,Xs).

consumer([X|Xs]) :-
   use(X),  consumer(Xs).

/* start consumer and producer in parallel */
main :- producer(1,S), consumer(S).
```

Figure 2. Implementation of streams with shared logical variables.

stream to the server. Such merge operations can be expressed in parallel logic languages [Shapiro and Safra 1986], but Gelernter argues that the resulting programs are less clear and concise than similar programs in languages supporting streams with multiple readers and writers.

2.2.3 Expressing and Controlling Nondeterminism

As discussed in Section 2.2, the interaction patterns between processes are not always deterministic, but sometimes depend on run-time conditions. For this reason, models for expressing and controlling nondeterminism have been introduced. Some communication primitives that we have already seen are nondeterministic. A message received indirectly through a port, for example, may have been sent by any process. Such primitives provide a way to *express* nondeterminism, but not to *control* it. Most programming languages use a separate construct for controlling nondeterminism. We will look at two such constructs: the *select statement*, used by many algorithmic languages, and the *guarded Horn clause*, used by most parallel logic programming languages. Both are based on the *guarded command statement*, introduced by Dijkstra as a sequential control structure [Dijkstra 1975].

The Select Statement. A select statement consists of a list of guarded commands of the following form:

guard → statements

The guard consists of a Boolean expression and some sort of "communication request." The Boolean expression must be free of side effects, as it may be evaluated more than once during the course of the select statement's execution. In CSP [Hoare 1978], for example, a guard may contain an explicit receipt of a message from a specific process *P*. Such a request may either *succeed* (if *P* has sent such a message), *fail* (if *P* has already terminated), or *suspend* (if *P* is still alive but has not sent the message yet). The guard itself can either succeed, fail, or suspend: The guard succeeds if the expression is "true" and the request succeeds; the guard fails if the Boolean expression evaluates to "false" or if the communication request fails; or the guard suspends if the expression is "true" and the request suspends. The select statement as a whole blocks until either all of its guards fail or some guards succeed. In the first case, the entire select statement fails and has no effect. In the latter case, one succeeding guard is chosen nondeterministically, and the corresponding statement part is executed.

In CSP, the select statement can be used to wait nondeterministically for specific messages from specific processes. The select statement contains a list of input requests and allows individual requests to be enabled or disabled dynamically. For example, the buffer process described above can interact with a consumer and a producer as shown in Figure 3. Communication takes place as soon as either (1) the buffer is not full and the producer sends a mes-

```
[
    not full(buffer); producer?DepositItem(x) →
          add x to end of buffer;

 [] not empty(buffer); consumer?AskForItem() →
          consumer!SendItem(first item of buffer);
          remove first item from buffer;
]
```

Figure 3. A select statement in CSP used by a buffer process. The statement consists of two guarded commands, separated by a "[ ]." The "?" is the input (receive) operator. The "!" is the output (send) operator.

sage, DepositItem; or (2) the buffer is not empty and the consumer sends a message, AskForItem. In the latter case, the buffer process responds by sending the item to the consumer.

CSP's select statement is asymmetric in that the guard in CSP can only contain an input operator, not an output operator. Thus, a process P can only wait to receive messages nondeterministically; it cannot wait nondeterministically until some other process is ready to accept a message from P [Bernstein 1980]. Output guards are excluded from most languages, because they usually complicate the implementation. Languages that do allow output guards include Joyce [Brinch Hansen 1987] and Pascal-m [Abramsky and Bornat 1983].

Select statements can also be used for controlling nondeterminism other than communication. Some languages allow a guard to contain a *time-out* instead of a communication request. A guard containing a time-out of T seconds succeeds if no other guard succeeds within T seconds. This mechanism sets a limit on the time a process will wait for a message. Another use of select statements is to control *termination* of processes. In Concurrent C, a guard may consist of the keyword **terminate**. A process that executes a select statement containing a **terminate** guard is willing to terminate if all other guards fail or suspend. If all processes are willing to terminate, the entire Concurrent C program terminates. Ada uses a similar mechanism to terminate parts of a program. Roughly, if all processes created by the same process are willing to terminate and the process that created them has finished the execution of its statements, all these processes are terminated. This mechanism presumes hierarchical processes.

A final note: Select statements in most languages are *unfair*. In the CSP model, for example, if several guards are successful, one of them is selected nondeterministically. No assumptions can be made about which guard is selected. Repeated execution of the select statement may select the same guard over and over again, even if there are other successful guards. An *implementation* may introduce a degree of fairness, by assuring that a successful guard will be selected within a finite number of iterations, or by giving guards equal chances. On the other hand, an implementation may evaluate the guards sequentially and always choose the first one yielding "true." The semantics of select statements do not guarantee any degree of fairness, so programmers cannot rely on it.

Proposals have been made for giving programmers explicit control over the selection of succeeding guards. Silberschatz suggests a partial ordering of the guards [Silberschatz 1984]. Elrad and Maymir-Ducharme propose prefixing every guarded command with a compile-time constant called the *preference control value* [Elrad and Maymir-Ducharme 1986]. If several guards succeed, the one with the highest preference control value (i.e., priority) is chosen. If there are several guards with this value, one of them is chosen nondeterministically. This feature is useful if some requests are more urgent than others. For example, the buffer process may wish to give consumers a higher priority than producers.

Guarded Horn Clauses. Logic programs are inherently nondeterministic. In reducing a goal of a logic program, there are often several clauses to choose from (see Section 2.1.1). Intuitively, the semantics of logic programming prescribe that the underlying execution machinery must simply choose the "right" clause, the one leading to a proof. This behavior is called *don't know nondeterminism.* In sequential logic languages (e.g., PROLOG), these semantics are implemented using *backtracking.* At each choice point, an arbitrary clause is chosen, and if it later turns out to be the wrong one, the system resets itself to the state before the choice point and then tries another clause.

In a parallel execution model, several goals may be tried simultaneously. In this model, backtracking is very complicated to implement. If a binding for a variable has to be undone, all processes that have used this binding must backtrack too. Most parallel logic programming languages therefore avoid backtracking altogether. Rather than trying the clauses for a given predicate one by one and backtracking on failure, parallel logic languages (1) search all these clauses in parallel and (2) do not allow any bindings made during these parallel executions to be visible to the outside until one of the parallel executions is committed to. This is called OR-parallelism (see Section 2.1.1). Unfortunately, this cannot go on indefinitely, because the number of search paths worked on in parallel will grow exponentially with the length of the proof.

A popular technique to control OR-parallelism is *committed-choice nondeterminism* (or *don't care nondeterminism*), which nondeterministically selects one alternative clause and discards the others. It is based on guarded Horn clauses of the following form:

$$A :- G_1, \ldots, G_n \mid B_1, \ldots, B_m, \qquad n \geqq 0, \quad m \geqq 0.$$

The conjunction of the goals G_i is called the *guard*; the conjunction of the goals B_i is the *body*. Declaratively, the *commit operator* " | " is also a conjunction operator.

Just like the guards of a select statement, the guard of a guarded Horn clause can either succeed, fail, or suspend. A guard suspends if it tries to bind a variable that it is not allowed to bind, as explained in Section 2.2.2. If a goal with a predicate A is to be reduced, the guards of all clauses for A are tried in parallel, until some guards succeed. The reduction process then chooses one of these guards nondeterministically and *commits* to its clause. It aborts execution of the other guards and executes the body of the selected clause.

So far, this all looks much like the select statement, but there are some subtle differences. A guard should not be allowed to affect its environment until it is selected. Guards that are aborted should have no side effects at all. Precautions must be taken against guards that try to bind variables in their environment. For example, consider the following piece of code:

$A(X)$:- $G(X) \mid B(X)$.
$A(X)$:- $H(X) \mid C(X)$.
$G(1)$:- $P(1)$.
$H(2)$:- $Q(2)$.

The guard G of the first clause binds X to 1 and then calls P. Guard H of the second clause binds X to 2 and calls Q. These bindings should not be made visible to the caller of A until one of the guards G or H is committed to. PARLOG ensures this by using mode-declarations to distinguish between input and output variables of a clause. The compiler checks that guards (or any other goals in the body) do not bind input variables. If a guard binds an output variable, this binding is initially made to a temporary variable. When a clause is committed to, the bindings made by its guard are made permanent, and the bindings generated by the other guards (to temporaries) are thrown away. If a guard is ultimately not selected, it has no effect at all.

Concurrent PROLOG, on the other hand, allows variables in the environment to be changed before commitment. But the effects only become visible outside the clause if the clause is committed to. The semantics and distributed implementation of commitment in Concurrent PROLOG are similar to those of atomic transactions [Taylor et al. 1987b].

Committed-choice nondeterminism has one severe drawback: It gives up completeness of search. In the code above, if $P(1)$, $C(2)$, and $Q(2)$ are true (i.e., can be proven), but $B(1)$ is false, then the goal $A(X)$ logically should succeed. The conjunction "$H(X)$ and $C(X)$" is true for $X = 2$ (recall that " | " is logically equivalent to "and"). As both guards $G(X)$ and $H(X)$ succeed, however, the system may select $G(X)$ and abandon the second clause. As $B(1)$ turns out to be false, the first clause will fail, and no solution will be found. The programmer must ensure that, at the time of commitment, either the right clause is selected or no clause resulting in a proof exists. This can be achieved by extending the guards to include $B(X)$ and $C(X)$:

```
A(X) :- G(X), B(X) | .
A(X) :- H(X), C(X) | .
```

This technique should not be used indiscriminately, because it restricts the effective parallelism. The binding to variable X is not made known to the caller of $A(X)$ until commitment, so in the new scheme, the caller will have to wait longer for this value to be available. This implies that, in general, guards should be kept as small as possible.

For reasons of simplicity and ease of implementation, most of the recent efforts in parallel logic programming languages center on their so-called "flat" subsets. In a flat guarded Horn clause, guards are restricted to simple predefined test predicates.

2.3 Partial Failure

The final issue that must be addressed by languages for programming distributed systems is the potential for partial failure of the system. Distributed computing systems have the potential advantages over centralized systems of higher *reliability* and *availability*. If some of the processors involved in a distributed computation crash, then, in principle, the computation can still continue on the remaining processors, provided that all vital information contained by the failing processors is also stored on some healthy ones. Thus, the system as a whole becomes more reliable. This principle of *replication* of information can be used to increase the availability of the system. A system is said to be *fault tolerant* if it still continues functioning properly in the face of processor crashes, allowing distributed programs to continue their execution and allowing users to keep on using the system.

In general, it is not an easy task to write programs that can survive processor crashes and continue as if nothing had happened. The responsibility for achieving reliability can be split up among the operating system, the language run-time system, and the programmer. Numerous research papers have been published about how operating systems can support fault tolerance [LeBlanc and Wilkes 1985; Powell and Presotto 1983]. In the following sections, we discuss how programming languages can contribute their part.

2.3.1 Programming Fault Tolerance

The simplest approach to handling processor failures is to ignore them altogether. This means that a single crash will cause the entire program to fail. Typically, processes trying to interact with a sick processor will either be blocked forever or discover an unexpected communication failure and terminate. A program running in parallel on several processors has a higher chance of failing than its single processor counterpart (although the shorter execution time of the parallel version may compensate a bit). Still, as processor crashes are rare, for many applications this is not a problem.

The next simplest approach to implementing fault tolerance is to let the programmer do it. The operating system or language run-time system can detect processor failures and return an error status to every process that wants to communicate with a crashed processor. The programmer can write code to deal with this contingency. For some programs, this approach is quite adequate. As an example, consider a distributed chess program in which each processor repeatedly chooses one possible move in the current board position, evaluates the new position, and returns some score. If a processor crashes before returning the result, all that need be done is to have another processor analyze the position. This simple scheme only works because the processors have no *side effects* except for returning the score. No harm is done if a position is examined twice.

A possible improvement to this scheme is to let the language run-time system take care of repeating requests to do some work. Nelson has studied this approach in the context of the Remote Procedure Call model [Nelson 1981]. If the run-time system detects that Processor *P* has crashed, *P*'s processes are restarted, either on *P* or on another processor. Furthermore, all outstanding RPCs to *P* are repeated.

As procedures can have side effects, it is important to specify accurately the semantics of a call that may have been executed (entirely or partially) more than once. Nelson gives a classification of these *call semantics*. The simplest case is a local procedure call (the caller and callee are on the same processor). If the processor does not crash, the call is executed exactly once (*exactly-once semantics*). If the processor does crash, the run-time system restarts all processes of the crashed processor, including the caller and the callee of the procedure. The call will eventually be repeated, until it succeeds without crashing. Clearly, the results of the last executed call are used by the caller, although earlier (abandoned) calls may have had side effects that survived the crash (e.g., changing a file in the processor's local disk). These semantics are called *last-one semantics.*

For RPCs, where the caller and callee are on different processors, the best that can be hoped for is to have the same semantics as for local calls, which are exactly-once semantics without crashes and last-one semantics with crashes. The former is not very hard to obtain, but achieving last-one semantics in the presence of crashes turns out to be tricky, especially if more than two processors are involved. Suppose Processor *P*1 calls Procedure *f* on Processor *P*2, which in turn calls Procedure *g* on Processor *P*3. While *P*3 is working on *g*, *P*2 crashes. *P*2's processes will be restarted, and P1's call to *f* will be repeated. The second invocation will again call procedure *g* on *P*3. Unfortunately, *P*3 does not know that *P*2 has crashed. *P*3 executes *g* twice and may return the results in any order, possibly violating last-one semantics. The problem is that, in a distributed environment, a crashed processor may still have outstanding calls to other processors. Such calls are appropriately called *orphans*, because their parents (callers) have died. To achieve last-one semantics, these orphans must be terminated before restarting the crashed processes. This can be implemented either by waiting for them to finish or by tracking them down and killing them ("orphan extermination"). As this is not an easy job, other (weaker) semantics have been proposed for RPC. *Last-of-many semantics* is obtained by neglecting orphans. It suffers from the problem described above. An even weaker form is *at-least-once semantics*, which just guarantees that the call is executed one or more times, but does not specify which results are returned to the caller.

One key idea is still missing from our discussion. Procedure calls (local as well as remote) can have side effects. If a call is executed many times (because of processor crashes), its side effects also are executed many times. For side effects like incrementing a bank account stored in a database, this may be highly undesirable (or highly desirable, depending on one's point of view). A mechanism is needed to specify that a call either runs to completion or has no effects at all. This is where *atomic transactions* come in.

2.3.2 Atomic Transactions

A distributed program can be regarded as a set of parallel processes performing operations on data objects. Usually, a data object is managed by a single process, but other processes can operate on the object indirectly (e.g., by issuing an RPC requesting the managing process to do the operation). In general, the effects of an operation become visible immediately. Moreover, operations affecting objects on secondary storage become permanent once the operation has been performed. Sometimes this behavior is undesirable. Consider a program that transfers a sum of money from one bank account (stored on disk) to another, by decreasing the first one and increasing the second one. This simple approach has two dangers. First, if another parallel process adds up all accounts in the database while the first process is in the middle of its transaction, it may observe the new value of the first account and the old value of the second, so that it uses inconsistent values. Second, if the process doing the transfer crashes immediately after decreasing the first account, it leaves the database in an inconsistent state. If it is restarted later, it may try to decrease the first account once more.

A solution to these problems is to group operations together in *atomic transactions* (also called *atomic actions* or simply *transactions*). A group of operations (called a transaction) is atomic if it has both the property of *indivisibility* and the property of *recoverability*. A transaction is indivisible if, viewed from the outside, it has no intermediate states. For the outside world (i.e., all other transactions), it looks as if either all or none of the operations have been executed. A transaction is recoverable if all objects involved can be restored to their initial state if the transaction fails (e.g., due to a processor crash), so that the transaction has no effect at all.

Recoverability can be achieved as follows: If a transaction contains an operation that tries to change an object, the changes are not applied to the original object, but to a new copy of the object, called a *version*. If the entire transaction fails (*aborts*), the new versions are simply discarded. If the transaction succeeds, it *commits* to these new versions. All objects changed by the transaction retain the value of their new version. Furthermore, the latest value of each object is also placed on *stable storage* [Lampson 1981], which has a very high chance of surviving processor crashes and is accessible by all processors.

Indivisibility can be trivially assured by executing all atomic transactions sequentially. In our bank account example, we could deny other processes access to the database while the first process is doing the transfer. Unfortunately, this severely limits parallelism and hence degrades performance. A more efficient approach is to synchronize processes by using finer-grained *locks*. The process doing the transfer first locks the two accounts. Other processes trying to access these two accounts are automatically suspended when they attempt to lock them.

Atomic transactions originated in the database world, but they are also used by some programming languages, such as Argus [Liskov 1988], Aeolus [Wilkes and LeBlanc 1986], and Avalon [Detlefs et al. 1988]. A programming language can provide convenient abstractions for data objects and invocations of atomic transactions. The language run-time system can take care of many details, like locking and version management. These issues are discussed in Section 3.1.7.

2.3.3 Transparent Fault Tolerance

The mechanisms discussed above provide linguistic support for dealing with partial failures. Some of the problems are solved by the operating system or the language run-time system, but programmers still have to do part of the work. This work has to be done for every new application. Other systems relieve programmers from all worries, by supporting fault tolerance in a fully transparent way.

Borg et al. describe a fault-tolerant message-passing system [Borg et al. 1983]. For each process, an inactive *backup* process is created on another processor. All messages sent to the primary process are also sent to its backup. The backup also counts the messages sent by the primary process. If the primary processor crashes, the backup process becomes active and starts repeating the primary process's computations. Whenever it wants to receive a message, the backup process reads the next message saved while the primary process was still alive. If the backup process needs to send a message, it first checks to see if the primary process had already sent it, to avoid sending messages twice. During normal computations, the primary and backup processes periodically synchronize, to copy the entire state of the primary process (a checkpoint) to the backup. The backup process then can forget all messages previous to the checkpoint.

This approach requires extra processors and will sometimes delay computation while a checkpoint is being made. Strom and Yemini propose a different technique, *optimistic recovery*, to be used in systems consisting of processes that interact only by message passing [Strom and Yemini 1985b]. (This model is used in their language NIL.) Their technique involves periodic checkpointing and logging of messages on stable storage, rather than to a backup process. As a fundamental departure from Borg's approach, these activities proceed *asynchronously* with the normal computations. This has the advantage that, if I/O bandwidth to stable storage is high enough, the normal computation will not slow down. However, the technique requires some bookkeeping overhead to allow a consistent system state to be restored after a crash.

3. LANGUAGES FOR PROGRAMMING DISTRIBUTED SYSTEMS

In this section we take a closer look at several languages that were designed for programming distributed systems. It is difficult to determine exactly how many such languages exist; we know of nearly 100 relevant languages, but there are probably many more. We have selected a subset for closer study. These languages together are representative of research in this area. We have chosen these languages to cover a broad spectrum of ideas. Although we have attempted to focus on languages that have been well documented and cited in the literature, we fully admit that any selection of this kind contains a certain amount of subjective choice. We include references to languages not discussed in detail here. An overview of the languages for distributed programming cited in this paper is given in the Appendix.

We have organized the languages in a simple classification scheme. First, we distinguish between *logically distributed* and *logically nondistributed* languages, as discussed in Section 1.3. In languages based on logical distribution, parallel computations (e.g., processes) communicate by sending messages to each other. The address spaces of different computations do not overlap, so the address space of the whole program is distributed. In a logically nondistributed language, the parallel units have a logically *shared* address space and communicate through data stored in the shared address space. Note that this distinction is based on the *logical model* of the language; the presence of logically shared data does not imply that *physical* shared memory is needed to implement the language. All languages described below that are based on logically shared data have been implemented on distributed computing systems, that is, on computers without shared primary memory.

The languages in the two categories are further partitioned into a number of classes, based on their communication mechanisms. In the first category, we include synchronous message passing, asynchronous message passing, rendezvous, RPC, multiple communication primitives, objects, and atomic transactions. In the second category, we distinguish between implicit communication through function-results (used in parallel functional languages), shared logical variables (parallel logic languages), and distributed data structures. The classification is illustrated in Figure 4.

In each of the following subsections, we discuss one class of languages. Each subsection starts with a table containing several languages of that class together with references to papers on these languages. Each table corresponds with one specific leaf in the tree of Figure 4. We have selected at least one language from each table for closer study. We describe the most distinctive features of the example language(s) and discuss how it differs from other members of its class. We emphasize the semantics, rather than the syntax. Our intention is to expose the new key ideas in the language, not to provide a full language description.

For each language, we first provide background information on its design. Next, we describe how parallelism is expressed in the

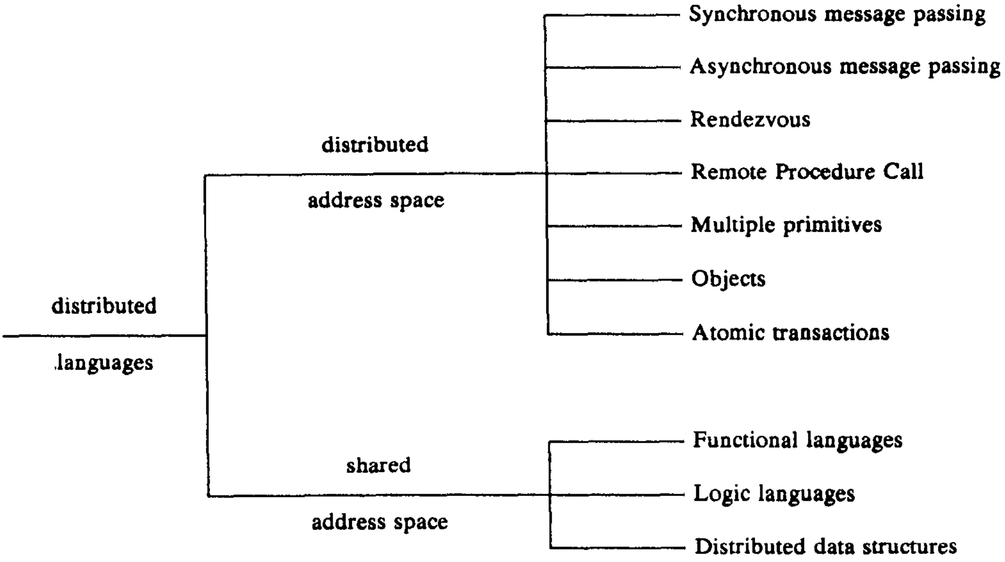

Figure 4. Classification of languages for distributed programming.

language and how parallel units are mapped onto processors (if the language addresses this issue). Subsequently, the communication and synchronization primitives are discussed. If relevant, we also discuss how the language deals with fault tolerance. Finally, we give information on implementations and user experiences with the language. Issues like support for distributed debugging and commercial availability of language implementations are outside the scope of this paper and are not discussed.

3.1 Languages with Logically Distributed Address Spaces

We discuss seven classes of languages with logically distributed address spaces: languages supporting synchronous message passing, asynchronous message passing, rendezvous, RPC, multiple communication primitives, operation invocations on objects, and atomic transactions. Languages in the first two classes provide point-to-point messages. Rendezvous-based languages support two-way communication between senders and receivers. An RPC is also a two-way interaction, but its semantics are closer to a normal procedure call. Languages in the fifth class use a variety of one-way and two-way (or even multiway) communication primitives. Object-based languages also support one or more of the above primitives. Unlike other languages, communication is between objects rather than processes. As objects encapsulate both data and behavior, these languages may also be thought of as providing some form of data sharing. Finally, we discuss languages based on atomic transactions; these languages are mainly intended for implementing fault-tolerant applications.

3.1.1 Synchronous Message Passing

In 1978 Hoare wrote what was later to become a very influential paper, although it described only a fragment of a language [Hoare 1978]. The language, called Communicating Sequential Processes (CSP), generated some criticism [Kieburtz and Silberschatz 1979; Bernstein 1980], but also stimulated the design of many other languages and systems (see Table 2). The CSP model consists of a fixed number of sequential processes that communicate only through synchronous message passing. Joyce differs from the other languages of Table 2 by supporting *recursive* processes. Below, we describe CSP in some detail and discuss one of its descendants, Occam.

Table 2. Languages Based on Synchronous Message Passing

Synchronous message passing	
Language	References
CCSP	[Hull and Donnan 1986]
CSM	[Zhongxiu and Xining 1987]
CSP	[Hoare 1978]
CSP-S	[Patniak and Badrinath 1984]
CSPS	[Roman et al. 1987]
CSP/80	[Jazayeri et al. 1980]
ECSP	[Baiardi et al. 1984]
GDPL	[Ng and Li 1984]
Joyce	[Brinch Hansen 1987]
LIMP	[Hunt 1979]
Occam	[Inmos Ltd. 1984]
Pascal-m	[Abramsky and Bornat 1983]
Pascal + CSP	[Adamo 1982]
Planet	[Crookes and Elder 1984]
RBCSP	[Roper and Barter 1981]

CSP. CSP was designed by Hoare as a simple language that allows an efficient implementation on a variety of architectures [Hoare 1978, 1985].[11]

(1) *Parallelism.* CSP provides a simple parallel command to create a fixed number of parallel processes. A process consists of a name, local variables, and a sequence of statements (body). CSP processes take no parameters and cannot be mapped onto specific processors. An array of similar processes can be created, but their number must be a compile-time constant. As a simple example of a parallel statement,

[writer :: x: real; . . . || reader(i: 1 . . 2) :: . . .]

creates three processes, called "writer," "reader(1)," and "reader(2)." The writer has a local variable x. The subscript variable i can be used within the body of the reader processes.

(2) *Communication and synchronization.* CSP processes may not communicate by using global variables. All interprocess communication is done using synchronous **receive** and **send**. The sending process specifies the name of the destination process and provides a value to be sent. The receiving process specifies the name of the sending process and provides a variable to which the received value is assigned. A process executing either a **send** or a **receive** is blocked until its partner has executed the complementary statement. Consider the following example:

[X :: Y ! 3 || Y :: n: integer; X ? n]

In Process X's statement, the value 3 is sent to Y. In Process Y's statement, input is read from Process X and stored in the local variable n. When both X and Y have executed their statements, the one-way communication occurs. The net result is assigning 3 to n.

Both simple and structured data may be communicated (and assigned), as long as the value sent is of the same type as the variable receiving it. The structured data can be given a name (a *constructor*), such as **pair** in the following example:

[X :: Y ! pair(35, 60) || Y :: n, m: integer;
  X ? pair(n, m)]

An empty constructor may be used to synchronize two processes without transferring any real data.

The *alternative* construction provides for nondeterminism in CSP. It consists of sets of guards followed by actions to be performed. The guards may contain Boolean expressions and an input statement, as explained in Section 2.2.3. CSP allows a process to receive selectively, based on the availability of input and the name field (constructor) of the incoming communication.

(3) *Implementation and experience.* CSP is essentially a paper design, but it has influenced the design of several languages (see Table 2) that have been implemented and used, most notably the Occam language.

Occam. Occam is modeled on Hoare's CSP and was designed for programming Inmos's transputer [Inmos Ltd. 1984; May 1983]. Occam is essentially the assembly language of the transputer. The language lacks features that have become standard in most modern programming languages,

[11] We describe the original language, outlined by Hoare [1978]; the 1985 version has a clearer syntax and uses named channels.

such as data typing, recursive procedures, and modules.

(1) *Parallelism.* There are three basic actions in Occam: *assignment, input,* and *output.* Each action is considered to be a little process. Processes can be grouped together in several ways to form more complex processes. Any process can be *named* by prefixing its definition with the keyword PROC, followed by its name and a list of formal parameters. When subsequently referenced, a new instance of the named process is created, with the parameters specified in the reference. Both parallel and sequential execution of a group of processes must be explicitly stated, by heading the group with a PAR or SEQ, respectively.

Arrays of similar processes can be expressed in Occam. In the construct

```
PAR i = 0 FOR n
  process . . .
```

n parallel processes are created, each with a different value for *i*.

Occam provides a facility for assigning processes to processors. Parallel processes may be prioritized by prefixing the group with PRI PAR. The first process in the group is given highest priority; the second, second highest priority; and so on.

(2) *Communication and synchronization.* Unlike CSP, parallel processes communicate indirectly through *channels.* A channel is a one-way link between two processes. Channel communication is fully synchronous. Only one process may be inputting from, and one outputting to, a channel at a given time. Channels are typed, and their names can be passed as parameters to PROC calls.

Occam provides an ALT construct, similar to CSP's alternative statement, to express nondeterminism. The constituents of this construct can be prioritized. If input is available on more than one channel, the one with the highest priority will be accepted.

The current time can be read from an input-only channel declared as a TIMER. A delay until a certain time can be made with the "WAIT AFTER *t*" construct. This can be used as a constituent of an ALT construct, for example, to prevent a process from hanging forever if no input is forthcoming.

(3) *Implementation and experience.* Occam was intended for use with multiple interconnected transputers, where a channel would be implemented as a link between two transputers [May and Shepherd 1984]. The transputer implementation is quite efficient (e.g., a context switch takes a few microseconds). This efficiency has been achieved by using a simple communication model (CSP) and by requiring the number of processes and their storage allocation to be determined at compile time. Occam has also been implemented on nontransputer systems [Fisher 1986].

Occam is used extensively for applications like signal processing, image processing, process control, simulation, and numerical analysis. A major criticism of the first version of Occam is the inability to pass complex objects (e.g., arrays) as part of a single message. Occam-2 has addressed this problem through the introduction of *channel protocols,* which describe the types of objects that may be transferred across a channel [Burns 1988]. The compiler (sometimes with the help of the run-time system) checks that the input and output operations on a channel are compatible with the channel protocol.

3.1.2 Asynchronous Message Passing

The synchronous message-passing model proposed by Hoare and adapted by Occam prevents the sending process from continuing immediately after sending the message. The sender must wait until the receiving process is willing to accept the message. This design decision has a major impact on both the programming style and the implementation of a language. Several language designers have chosen to remove this restriction and support *asynchronous* message passing, sometimes in addition to synchronous message passing. Languages in this class are shown in Table 3. We discuss NIL in more detail.

Table 3. Languages Based on Asynchronous Message Passing

Asynchronous message passing	
Language	References
AMPL	[Dannenberg 1981; Milewski 1984]
CMAY	[Bagrodia and Chandy 1985]
Concurrent C	[Tsujino et al. 1984]
CONIC	[Kramer and Magee 1985; Sloman and Kramer 1987]
DPL-82	[Ericson 1982]
FRANK	[Graham 1985]
GYPSY	[Ambler et al. 1977]
LADY	[Nehmer et al. 1987]
MENYMA/S	[Koch and Maibaum 1982]
NIL	[Strom and Yemini 1983]
ParMod	[Eichholz 1987]
PCL	[Lesser et al. 1979]
Platon	[Staunstrup 1982]
PLITS	[Feldman 1979]
Port Language	[Kerridge and Simpson 1986]
Pronet	[LeBlanc and Maccabe 1982]
ZENO	[Ball et al. 1979]

NIL. NIL (Network Implementation Language) is a high-level language for the construction of large, reliable, distributed software systems [Strom and Yemini 1983, 1984, 1985a, 1986]. NIL was designed by Robert Strom and Shaula Yemini at the IBM T. J. Watson Research Center.

NIL is a *secure* language, which means that one program module cannot affect the correctness of other modules (e.g., by a "wild store" through a bad pointer). The importance of security is pointed out by Hoare [1981]. Security in NIL is based on an invention called the *typestate* [Strom and Yemini 1986]. A typestate is a compile-time property that captures both the type of a variable and its state of initialization. In the program fragment

1. *X*, *Y*: INTEGER;
2. **if** condition **then** *X* := 4; **end if**
3. *Y* := *X* + 3;

statement 3 is marked as illegal by the compiler, because variable *X* might still be uninitialized at this point. *X* has the right type (integer), but the wrong state. The typestate mechanism imposes some constraints on the structure of the programs (especially on the control flow), but the designers claim that these constraints are not overly restrictive and usually lead to better structured code. NIL avoids features that would make compile-time checking of typestates impossible. It does not provide explicit pointer manipulation (it does provide a higher level construct for building general data structures), and it has an IPC model that disallows sharing of variables.

(1) *Parallelism.* Parallelism in NIL is based on the so-called *process model* [Strom et al. 1985; Strom 1986]. A NIL system consists of a network of dynamically created processes that communicate only by message passing over communication channels. In NIL, a process is not only the unit of *parallelism*, but also the unit of *modularity*. The division of a NIL program into processes should be based on software engineering principles rather than on performance considerations. The mapping of processes onto processors is considered to be an implementation issue, to be dealt with by the compiler and run-time system. This process model makes NIL conceptually simpler than languages that have separate mechanisms for parallelism and modularity (e.g., tasks and packages in Ada).

(2) *Communication and synchronization.* Configuration of the communication paths between processes is done dynamically. A *port* in NIL is a queued communication channel. At a given time, a port has one

specific owner. Ownership of a port can be transferred to another process, by passing the port as part of a message or by passing the port as an initialization parameter to a newly created process. A process can connect input ports and output ports owned by it.

Both synchronous communication and asynchronous communication are supported. A single input port may be connected to several output ports, so there can be multiple pending messages on an input port; these messages therefore have to be queued. A guarded-command style statement is provided for waiting for messages on any of a set of input ports.

(3) *Fault tolerance.* Recovery from processor failures is intended to be handled transparently by the NIL run-time system, using the optimistic recovery technique discussed in Section 2.3.3.

(4) *Implementation and experience.* A NIL compiler generating code for a uniprocessor (IBM 370) has been implemented. Research on distributed implementations has focused on transformation strategies, which optimize NIL programs for specific target configurations [Strom and Yemini 1985a]. NIL has been used to implement a prototype communication system, consisting of several hundred modules [Strom and Yemini 1986]. The implementors found the typestate mechanism highly useful in integrating this relatively large number of modules.

3.1.3 Rendezvous

The rendezvous mechanism was first used in Ada and later employed in some other languages, as shown in Table 4. We discuss Ada and Concurrent C below.

Table 4. Languages Based on Rendezvous

Rendezvous	
Language	References
Ada	[U.S. Department of Defense 1983]
BNR Pascal	[Gammage et al. 1987]
Concurrent C	[Gehani and Roome 1989]
MC	[Rizk and Halsall 1987]

Ada. The language Ada was designed on behalf of the Department of Defense by a team of people led by Jean Ichbiah [U.S. Department of Defense 1983]. Since its first (preliminary) definition appeared in 1979, Ada has been the subject of an avalanche of publications. A substantial part of the discussion in these publications relates to parallel and distributed programming in Ada (e.g., [Yemini 1982; Gehani 1984a; Mundie and Fisher 1986; Burns et al. 1987]) and to the implementation of Ada's multitasking. van Katwijk reviews more than 30 papers of the latter category [van Katwijk 1987].

(1) *Parallelism.* Parallelism is based on sequential processes, called *tasks* in Ada. Each task has a certain type, called its *task type.* A task consists of a *specification* part, which describes how other tasks can communicate with it, and a *body,* which contains its executable statements. Tasks can be created explicitly or can be declared, but in neither case is it possible to pass any parameters to the new task. Limited control over the local scheduling of tasks is given, by allowing a static priority to be assigned to task types. There is no notation for mapping tasks onto processors.

(2) *Communication and synchronization.* Tasks usually communicate through the rendezvous mechanism. Tasks can also communicate through shared variables, but updates of a shared variable by one task are not guaranteed to be immediately visible to other tasks. An implementation that does not have physically shared memory may keep local copies of shared variables and defer updates until tasks explicitly synchronize through a rendezvous.

The rendezvous mechanism is based on entry declarations, entry calls, and accept statements, as discussed in Section 2.2.1. Entry declarations are only allowed in the specification part of a task. Accept statements for the entries appear in the body of the task. They contain a formal parameter part similar to that of a procedure. It is not possible to accept an entry conditionally depending on the values of the actual parameters, or to control the order in which outstanding requests are accepted. Gehani and Cargill show that an array of entries with the same formal part (a so-called *family*) can sometimes be used instead of conditional acceptance, although in general this leads to polling [Gehani and Cargill 1984].

A task can call an entry of another task by using an *entry call statement*, similar to a procedure call statement. An entry call specifies the name of the task containing the entry, as well as the entry name itself. Entry names cannot be used in expressions (e.g., they cannot be passed around as parameters). A program can use a pointer to an explicitly created task as a name for that task. Pointers are more flexible than static identifiers, but they cannot point to declared tasks or to entries.

Ada uses a **select** statement similar to CSP's alternative command for expressing nondeterminism. Ada's **select** statement is actually used for three different purposes: to select an entry call nondeterministically from a set of outstanding requests, to call an entry conditionally (i.e., only if the called task is ready to accept it immediately), and to set a time-out on an entry call. So Ada essentially supports input guards and conditional and timed entry calls, but not output guards.

(3) *Fault tolerance.* Ada has an exception-handling mechanism for dealing with software failures, but the language definition does not address the issue of hardware failures [Burns et al. 1987]. If the processor on which a task T executes crashes, an implementation may (but need not) treat T like an aborted task (i.e., a task that failed because of software errors). If so, other tasks that try to communicate with T will receive a **tasking_error** exception and conclude that T is no longer alive; however, they do not know the reason (hardware or software) why T died, so this support for dealing with processor failures is very rudimentary.

(4) *Implementation and experience.* Given the fact that the Department of Defense intends to have Ada replace 300 or so other languages currently in use and that industry has also shown some interest in Ada, the language probably will be used extensively in the future. Many implementations of Ada are now available, and several million lines of Ada code have already been written for uniprocessor applications [Myers 1987].

Burns et al. cite 18 papers addressing the issue of how to use Ada in a distributed environment [Burns et al. 1987]. They also review many problems with parallel and distributed programming in Ada. The synchronization mechanism receives a substantial part of the criticism: It is asymmetric (input guards but not output guards in **select** statements), entry calls are always serviced in FIFO order and cannot be accepted conditionally, and it is not possible to assign priorities to alternatives of a **select** statement. Distribution of programs among multiple processors is not addressed by the definition of Ada, but is left to configuration tools.

Concurrent C. Concurrent C extends the C language [Kernighan and Ritchie 1978] by adding support for distributed programming. The language is being developed at AT&T Bell Laboratories, by Narain Gehani and others [Gehani and Roome 1986a, 1989]. Concurrent C is based on Ada's rendezvous model, but its designers tried to avoid the problems they observed in this model [Gehani and Roome 1988].

(1) *Parallelism.* A process in Concurrent C has a specification part and a body, just like tasks in Ada. The specification part consists of the process's name, a list of formal parameters, and a list of *transactions.* (A transaction is Concurrent C's equivalent to an Ada entry.) Processes are created explicitly, using the **create** primitive, which can pass parameters to the created process. The new process can be given a priority (which can later be changed by itself or by other processes) and can be assigned to a specific processor. The **create** primitive returns an identifier for the new process instantiation. This value can be assigned to a variable of the same process type and can be passed around as a parameter. For example,

```
process buffer pid;
pid = create buffer(100) priority(1)
    processor(3);
```

starts a process of type *buffer* on processor 3, giving it priority 1 and passing the number 100 as a parameter to it. A reference to the process is returned in *pid*, which might be passed to another process to use for subsequent communication with the buffer process.

(2) *Communication and synchronization.* Processes communicate through the rendezvous mechanism. (Communication through shared variables is not forbidden, but no special language support is provided for it, and it will only work correctly on shared-memory machines.) A transaction in Concurrent C differs from an Ada entry in that a transaction may return a value. In addition, Concurrent C supports *asynchronous* transactions (equivalent to asynchronous message passing); such transactions may not return a value [Gehani 1987].

Concurrent C supports a more powerful **accept** statement than Ada. Transactions can be accepted conditionally, based on the values of their parameters, as in the following example:

accept tname(*a*, *b*, *c*) **suchthat** ($a < b$)
  {. . .}

Only outstanding transaction calls for which the expression after the **suchthat** evaluates to "true" will be accepted. The order of acceptance can be controlled using a **by** clause:

accept tname(*a*, *b*, *c*) **by**(*c*) {. . .}

Of all outstanding calls to transaction tname, the one with the lowest third parameter will be accepted.

A transaction call is similar to a function call and can be used as part of an expression (since transaction calls may return values). The transaction call specifies the name of the called process along with the transaction name and supplies actual parameters. The process name can be any expression that yields a process identifier. The transaction name is a static identifier. A specific transaction of a specific process can be assigned to a *transaction pointer* variable, which can subsequently be used instead of these two names in an indirect transaction call. With this mechanism, the caller need not know the type or name of the called process.

The caller can specify the amount of time it is willing to wait for its request to be carried out, using the following construct:

within *N* ? pid.tname(params) : expr

If process *pid* does not accept the *tname* transaction call within *N* seconds, the call is canceled, and the expression *expr* is evaluated instead. This construct is equivalent to Ada's timed entry call, although with an entirely different syntax.

Nondeterminism is expressed through a **select** statement, similar to the one used by Ada. Concurrent C's select statement is somewhat cleaner, because it is used only for dealing with nondeterminism, not for timed or conditional transaction calls.

(3) *Fault tolerance.* A fault-tolerant version of Concurrent C (called FT Concurrent C) based on replication of processes has been designed [Cmelik et al. 1987].

(4) *Implementation and experience.* Concurrent C has been implemented on a uniprocessor, a group of executable-code-compatible machines connected by an Ethernet, and a multiprocessor providing shared global memory [Cmelik et al. 1986].

Concurrent C has been used in several nontrivial applications, such as a distributed version of "make" [Cmelik 1986], a robot system [Cox and Gehani 1986], discrete event simulation [Roome 1986], and a window manager [Smith-Thomas 1986]. The language is being merged with C++ [Stroustrup 1986] to create a programming language supporting both distributed programming and classes [Gehani and Roome 1986b].

3.1.4 Remote Procedure Call

RPC was first introduced by Brinch Hansen for his language Distributed Processes (see below) and has been studied in more detail by Nelson and Birrell [Nelson 1981; Birrell and Nelson 1984]. Remote procedure calls are also used in several other languages, as shown in Table 5. (Most languages based on atomic transactions also use RPCs; these are discussed in Section 3.1.7.)

Table 5. Languages Based on Remote Procedure Call

Remote procedure call	
Language	References
Cedar	[Swinehart et al. 1985]
Concurrent CLU	[Hamilton 1984; Cooper and Hamilton 1988]
Distributed Processes	[Brinch Hansen 1978]
LYNX	[Scott 1985, 1986, 1987; Scott and Cox 1987]
P$^+$	[Carpenter and Cailliau 1984]

Distributed Processes. Brinch Hansen's Distributed Processes (DP) [Brinch Hansen 1978] is the successor to Concurrent Pascal [Brinch Hansen 1975]. Like Concurrent Pascal, DP is oriented toward real-time systems programming. Instead of Concurrent Pascal's monitor-based communication scheme, DP processes communicate using RPC.

(1) *Parallelism.* In DP the number of processes is fixed at compile time. The intention is that there be one processor dedicated to executing each process. Each process, however, can contain several threads of control running in pseudoparallel. A process definition contains an *initial* statement, which may be empty; this is the first thread. It may continue forever, or it may finish executing at some point, but in either case the process itself continues to exist; DP processes never terminate. Additional threads are initiated by calls from other processes. Arrays of processes may be declared. A process can determine its array index using the built-in function **this**.

(2) *Communication and synchronization.* DP processes communicate by calling one another's *common procedures.* Such a call has the form

call *P.f* (exprs, vars)

where *P* is the name of the called process and *f* is the name of a procedure declared by *P*. The expressions are input parameters; the return values of the call are assigned to the (output) variables.

The calling process (and all its threads) is blocked during the call. A new thread of control is created within *P*. *P*'s *initial* statement and the threads created to handle remote calls execute as pseudoparallel processes, scheduled nonpreemptively. They communicate through *P*'s global variables and synchronize through guarded regions.

Like the select statement of CSP and Ada, a guarded region in DP is based on Dijkstra's guarded command. A guarded region allows a thread to wait until one of a number of guards (conditional expressions) is true. When a thread is blocked in a guarded region, other threads in its process can continue their execution. The guards have access to the input parameters of the remote call and to the process's global variables. Since other threads can change the global variables, the guards are repeatedly evaluated, until one or more of them is true. This is a major difference with the select statement and makes the guarded region somewhat more powerful.

Two forms of guarded regions are supported by DP. The **when** statement nondeterministically selects one true guard and executes the corresponding statement. The **cycle** statement is an endless repetition of the **when** statement.

(3) *Implementation and experience.* DP is a paper design and has not been implemented. An outline of a possible implementation is given by Brinch Hansen [1978].

3.1.5 Multiple Communication Primitives

As can be seen from the previous sections, many different communication and synchronization mechanisms exist, each with its own advantages and disadvantages. As there is no general agreement on which primitive is best, some language designers have taken the approach of providing a range of primitives, from which the programmer can choose the one most suited to the application. In addition, programmers can experiment with different primitives while still using the same language. An important issue in the design of such a language is how to integrate all these primitives in a clean and consistent way. Examples of languages in this class are shown in Table 6. We discuss SR below.

Table 6. Languages Based on Multiple Communication Primitives

Multiple communication primitives	
Language	References
Dislang	[Li and Liu 1981]
Pascal-FC	[Burns and Davies 1988]
StarMod	[Cook 1980; LeBlanc and Cook 1983]
SR	[Andrews 1981]

Synchronizing Resources. Synchronizing Resources (SR) was developed by Gregory Andrews et al. at the University of Arizona [Andrews 1981, 1982; Andrews and Olsson 1986; Andrews et al. 1988]. SR is a language for programming distributed operating systems and applications. It is based on Modula, Pascal, and DP, and provides several models of interprocess communication.

(1) *Parallelism.* An SR program consists of one or more *resources.* A resource is a module run on one physical node (either a single processor or a shared-memory multiprocessor). Resources are dynamically created (parameters may be passed), and optionally assigned to run on a specific machine. An identifier for the resource instance is returned by the **create** command.

A resource can contain several processes, and these may share data. Synchronization among these processes is supported by the use of semaphores. Communication with processes in other resources is restricted to *operations,* discussed below. A resource may contain an initialization and a termination process. These are created and run implicitly. A resource terminates when it is killed by the **destroy** command. A program terminates when all its processes terminate or block.

(2) *Communication and synchronization.* An SR operation definition looks like a procedure definition. Its implementation can look like either a procedure or an entry point. When implemented as a procedure, the operation is serviced by an implicitly created process. When implemented as an entry point, it is serviced by an already running process in a rendezvous. The two types of implementation are transparent to the invoker of the operation. On the invoker's side, an operation may be called asynchronously using a **send** or synchronously using a **call.** A **send** blocks until the message has been delivered to the remote machine; a **call** blocks until the operation has been completed and any return values have been received. Several **calls** can be grouped in a parallel call statement, which terminates when all calls have been completed. The operation and its resource instance must be named explicitly in the invocation. This is done using the identifier for the resource returned by the **create** command.

By combining the two modes of servicing operations and the two modes of invoking them, four types of interprocess communication can be expressed as shown in Table 7.

SR uses a construct similar to the **select** statement (see Section 2.2.3) to deal with nondeterminism. The SR guarded command, or **alternative**, has the following form:

```
entry_point(params) and bool-expr
    by expr → statements
```

A guard may contain an entry point for an operation, a Boolean expression, and a priority expression. The two expressions can refer to the actual parameters of the operation. An alternative is enabled if there is a pending invocation of the operation and the Boolean expression evaluates to true. The expression in the **by** part is used for priorization when there are several pending invocations of the same operation. If all Boolean expressions are false, the process suspends.

(3) *Fault tolerance.* SR supports two rudimentary mechanisms for handling failures [Andrews et al. 1988]. Exception handlers can be used to handle failures detected by the run-time system. For example, a handler attached to an operation invocation is called if the invocation fails. A **when**-statement can be used to ask the run-time system to monitor a certain source (e.g., a process or processor) and to invoke a user-supplied operation if the source fails.

Table 7. Four Types of Interprocess Communication in SR

	Call (synchronous)	**Send** (asynchronous)
Entry (synchronous)	Rendezvous	Message passing
Process (asynchronous)	RPC	Fork

(4) *Implementation and experience.* An implementation of SR on top of UNIX[12] is described by Andrews et al. [1988]. It runs on collections of SUNs or VAXes and on the Encore Multimax. SR has been used to implement a parallel PROLOG interpreter and the file system of the Saguaro distributed operating system [Andrews et al. 1988].

3.1.6 Object-Based Languages

The object-based approach to programming is becoming increasingly popular, not only in the world of sequential programs, but also for building distributed applications. The need for distributed objects arises when, for example, operating systems and distributed problem solvers are modeled. Exploiting parallelism to speed up programs is usually considered to be a secondary issue, to be dealt with by the language implementation.

In most parallel object-based or object-oriented[13] languages (see Table 8), parallelism is based on assigning a parallel process to each object, so objects become active components. This method is used, for example, in the languages Concurrent Smalltalk, CLIX, Emerald, Hybrid, Ondine, POOL, Sloop, and in the actor languages Act 1, Cantor, and CSSA. Concurrent Smalltalk (based on Smalltalk-80 [Goldberg and Robson 1983]) also supports asynchronous message passing to increase parallelism even further. Actor languages [Hewitt 1977; Agha 1986] are related to object-oriented languages, but arrange objects in dynamically changing hierarchies, rather than in static classes. Athas and Seitz discuss the usage of the actor language Cantor for programming fine-grain multicomputers [Athas and Seitz 1988].

ABCL/1 uses asynchronous message passing and an explicit construct for sending several messages simultaneously to different objects. Orient84/K is a multiparadigm language for programming knowledge-based systems, integrating object-oriented, logic, and parallel programming. OIL is the intermediate language for the FAIM-1 symbolic multiprocessor system. OIL integrates parallel, object-oriented, logic, and procedural programming. Raddle is a language for the design of large distributed systems. EPL is an object-based language based on Concurrent Euclid [Holt 1982]. EPL is used with the Eden distributed operating system [Almes et al. 1985]. It influenced the design of Emerald (discussed below) and Distributed Smalltalk, which is based on Smalltalk-80.

Emerald. Emerald is an object-based programming language for the implementation of distributed applications. It was developed at the University of Washington by Andrew Black and others [Black et al. 1986, 1987; Hutchinson 1987; Jul et al. 1988; Jul 1988].

Like Smalltalk-80, Emerald considers all entities to be objects. For the programmer, both a file accessible by many processes and a Boolean variable local to a single process are objects. Objects can either be passive (data) or active. Unlike Smalltalk-80, Emerald is a strongly typed language and has no classes or inheritance.

Abstract types are used to define the interface to an object. An abstract type can be implemented by any object supporting at least the operations specified in the interface. The type system was designed to allow multiple implementations of the same type to coexist; new implementations can be added to a running system. The pro-

[12] UNIX is a registered trademark of AT&T Bell Laboratories.

[13] As discussed in Section 2.1.1, we define a language to be object oriented if it has inheritance, and object based if it supports objects but lacks inheritance.

Table 8. Object-Oriented, Object-Based, and Actor Languages

Objects	
Language	References
ABCL/1	[Yonezawa et al. 1986]
Act 1	[Lieberman 1987]
ALPS	[Vishnubhotia 1988]
Cantor	[Athas and Seitz 1988]
CLIX	[Hur and Chon 1987]
Cluster 86	[Lujun and Zhongxiu 1987]
ConcurrentSmalltalk	[Yokote and Tokoro 1986, 1987a, 1987b]
CSSA	[Nehmer et al. 1987]
Distributed Smalltalk	[Bennett 1987]
Emerald	[Black et al. 1986]
EPL	[Black et al. 1984]
Hybrid	[Nierstrasz 1987]
Mentat	[Grimshaw and Liu 1987]
OIL	[Davis and Robison 1985]
Ondine	[Ogihara et al. 1986]
Orient84/K	[Ishikawa and Tokoro 1987]
POOL	[America 1987]
Raddle	[Forman 1986]
SINA	[Aksit and Tripathi 1988]
Sloop	[Lucco 1987]

grammer can supply different implementations, each tailored to a specific use. Alternatively, the compiler can automatically generate different implementations from the same source code, tailored for local objects or distributed objects.

(1) *Parallelism.* Parallelism is based on the simultaneous execution of active objects. Since objects in Emerald can be moved from one processor to another (as discussed below), the language essentially supports process migration. This is the most flexible mapping strategy discussed in Section 2.1.2.

(2) *Communication and synchronization.* An object consists of four parts: a *name*, a *representation*, a set of *operations*, and an optional *process*. The name uniquely identifies the object within the distributed system. The representation contains the data of the object. Objects communicate by invoking each other's operations. There can be multiple active invocations within one object. The optional process runs in parallel with all these invocations. The invocations and the data shared by these invocations can be encapsulated in a monitor construct. The internal process can enter the monitor by calling an operation of its own object. Within a distributed system, many objects can run in parallel. Emerald provides the same semantics for local and remote invocations.

At any given time, an object is on a single processor, called its *location.* In general, programmers do not have to worry about locations, because the semantics of operation invocations are location independent. Some applications, however, are better off when they can control the locations of objects. For example, two objects that communicate frequently can be put on the same processor. Conversely, objects that are replicas of the same data should be located on different processors, to reduce the chance of losing the data after a processor crash.

In Emerald, global objects can be moved from one processor to another. Such a move may be initiated either by the compiler (using compile-time analysis) or by the programmer, using a few simple language primitives. One important case is where an object is passed as a parameter in a remote operation. Every access to the parameter object will result in an extra remote invocation. The obvious solution is to pass a *copy* of the object as a parameter, but this changes the parameter mechanism into call-by-value. For object-based languages, call-by-reference is more natural. The solution in Emerald is to optimize such calls by first moving the parameter object to the destination processor, then doing the call, and optionally moving the object back. As this case occurs frequently, Emerald introduces a new parameter passing mode, called *call-by-move*, to accomplish this efficiently (i.e., with low message-passing overhead).

(3) *Implementation and experience.* An important goal of a good implementation of Emerald is to recognize simple operations on small objects and to treat them efficiently. For example, an addition of two local integer variables is compiled to inline code. Local calls to objects that cannot move essentially take a local procedure call. Global objects (which are allowed to move) are referenced indirectly through an *object descriptor*, which either contains the address of the object if it is local, or tells where to find the object in the distributed system. When an object moves to another processor, the descriptors on its old and new locations are updated.

A prototype distributed implementation of Emerald has been built, running on DEC MicroVax II and SUN workstations connected by Ethernet.

Emerald has been used to implement a mail system, a replicated name server, a shared appointment calendar system, and several other applications [Jul et al. 1988].

3.1.7 Atomic Transactions

Several languages that were specifically designed for building fault-tolerant applications support atomic transactions in combination with RPC (see Table 9). Aeolus and Avalon are built on top of existing systems that already support atomic transactions. Aeolus provides language support for the Clouds operating system [LeBlanc and Wilkes 1985]. Avalon is being implemented on top of the Camelot distributed transaction management system [Spector et al. 1986]. Camelot applications can also use the Camelot Library, which is a collection of macros and subroutines embedded in the C language. The Camelot Library takes care of many low-level details of transaction and object management, thus facilitating the implementation of Camelot servers and applications. This approach avoids designing a totally new language, while providing higher level primitives than traditional system calls.

Table 9. Languages Based on Atomic Transactions

Atomic transactions	
Language	References
Aeolus	[Wilkes and LeBlanc 1986]
Argus	[Liskov and Scheifler 1983]
Avalon	[Detlefs et al. 1988]
Camelot Library	[Bloch 1988]

Argus. Argus [Liskov 1982; Liskov and Scheifler 1983; Liskov 1984, 1988; Weihl and Liskov 1985], being developed at MIT by Barbara Liskov and colleagues, is based on CLU [Liskov et al. 1977] and Extended CLU [Liskov 1979]. It provides support for fault-tolerant distributed programming, in particular for applications requiring a high degree of reliability and availability, such as banking, airline reservation, and mail systems. Its main features are *guardians* (modules that can survive crashes) and *actions* (groups of atomic executions).

(1) *Parallelism.* An Argus module, called a *guardian*, contains data objects and procedures for manipulating those objects. A guardian may contain **background** and **recover** sections and may have several **creator** and **handler** procedures. A creator procedure is run when an instance of the guardian is being made. The handler procedures are run on behalf of processes outside of the guardian. The recover section is executed when the guardian is started up again after a crash. The background section is intended for doing periodic tasks and is run continually during the life of the guardian.

A guardian instance is created dynamically by a call to a creator procedure. A creator may take parameters, and the guardian may be explicitly placed on a node:

```
guardianType$creator(params)
    @ machineX
```

More than one process may be running in a guardian instance at a given time. If the guardian contains a background section, a process is created to run it. In addition, each time a call is made to one of the guardian's handlers, a process is created to run the appropriate handler procedure. A guardian can terminate itself by executing the **terminate** statement.

Parallelism results from simultaneous execution of guardians. Pseudoparallelism results from the implicit creation of a new process for each handler call within a guardian. Pseudoparallel execution can also be expressed using a **coenter** statement. A **coenter** terminates when all its components have finished, and one component may terminate the rest prematurely by transferring control out of the **coenter** statement.

(2) *Communication and synchronization.* Processes running in the same guardian instance can communicate using shared variables. Processes belonging to different guardians, however, can only communicate using *handler calls.* A handler call is a form of RPC, with arguments passed by value. Guardian and handler names may be passed as parameters. Argus provides synchronization mechanisms at two levels: one for pseudoparallel processes; the other for parallel actions.

The **mutex** type provides mutually exclusive access to objects shared by processes within a guardian. The **seize** construct delays a process until it can gain possession of the given mutex object; the process gives up possession again when it finishes executing the seize body. The **pause** call can be made when a process encounters a delay (such as an unavailable resource) and wants to give up the mutex object while it suspends for a while.

In order to allow parallelism of actions, while retaining their atomic semantics, *atomic objects* are used, which are instances of atomic data types. Argus has some built-in atomic types, and users can define their own. The types of atomic objects determine the amount of parallelism of actions permitted.

(3) *Fault tolerance.* Some of the guardian's objects may be declared as **stable**; they are kept on stable storage. If a node crashes, the guardian can be brought up again by retrieving its stable objects from store and executing its recover section.

Argus supports two types of atomic executions: *topactions* and *nested subactions* [Moss 1981]. Changes only become permanent (and stable objects written back to stable storage) when a topaction commits. A subaction is indivisible, but its effects are not made permanent until its top-level action commits. If a top-level action aborts, its subactions have no effect at all. On the other hand, if a subaction aborts, its parent action is not forced to abort. Nested subactions can be used for dealing with communication failures and for increasing parallelism. An action can also start up a new topaction.

(4) *Implementation and experience.* A UNIX-based implementation of Argus on a collection of MicroVax II workstations is described by Liskov et al. [1987].

One Argus application reported in the literature is a distributed collaborative editing system (CES), which allows a group of coauthors to edit a shared document [Greif et al. 1986]. A number of problems with the language design were identified during this experiment. When an action aborts, for example, no user code is activated; the run-time system does all the processing automatically. In some cases, however, the application also needs to do some processing after an abort (e.g., in CES an abort sometimes implies updating a screen). The implementors of CES reported that their task would have been significantly simplified had Argus provided more explicit control over action aborts and commits.

Another application implemented in Argus, a distributed mail repository, is described by Day [1987].

Aeolus. Aeolus is a systems programming language for implementing fault-tolerant servers for the Clouds distributed operating system. Aeolus provides abstractions for the Clouds notions of *objects*, *actions*, and *processes*, and provides access to the recoverability and synchronization features supported by Clouds. Both Clouds and Aeolus are being developed at Georgia Institute of Technology by Richard LeBlanc and colleagues [LeBlanc and Wilkes 1985; Wilkes and LeBlanc 1986, 1988].

(1) *Parallelism.* Aeolus is object based in the sense that it supports data abstractions. Unlike in Emerald, however, objects in Aeolus are *passive* (see Section 2.1.1). Aeolus therefore supports a *process* concept for providing parallel activity.

(2) *Communication and synchronization.* Communication and synchronization in Aeolus are expressed through operation invocations on objects, as discussed below.

(3) *Fault tolerance.* We first give a brief description of salient features of Clouds related to fault tolerance and then discuss how Aeolus supports these features. The Clouds distributed operating system supports atomic transactions on *objects.* As in Argus, actions can be nested. A Clouds object is a passive entity that encapsulates data. The data of an object can only be manipulated by invoking operations (remote procedures) defined for the object. Objects are created dynamically. Each instance of an object type has its own *state*, consisting of the global variables used by the implementation of the operations. Objects may be replicated in order to increase availability [Wilkes and LeBlanc 1988].

Clouds supports so-called *recoverability* and *synchronization* of objects. Recoverability allows objects to survive processor crashes. Synchronization ensures that parallel operation invocations are ordered such that they do not interfere with each other. Both features can be handled automatically by the Clouds kernel or can be custom programmed for higher efficiency, using semantic knowledge about the problem being implemented. Automatic recovery is based on checkpointing the entire state of the object, whereas custom recovery need only checkpoint those parts of the object state that have been indicated by the programmer. Automatic synchronization allows multiple read-operations to execute simultaneously, but serializes all operations that modify any part of the object state.

Aeolus gives programming language support for Clouds objects and actions. An object type in Aeolus consists of a *definition part* and an *implementation part.* The former contains the name of the object type and the operations allowed on objects of that type. It also specifies whether recovery should be done by the system, by the programmer, or not at all, and it may specify that synchronization is to be done automatically. Programmed synchronization is based on critical sections (for mutual exclusion) and on explicit locking using various *lock types.* The declaration of a lock type specifies a number of *modes*, a *compatibility* relation between the nodes, and (optionally) a *domain* of values to be locked. For example, a read-write lock type over file names of 14 characters can be declared as follows:

```
type rw_lock is
  lock (read: [read], write: [ ])
    domain is string(14)
```

This declaration introduces two modes, "read" and "write," and specifies that several "read" locks on file names may be obtained, but that "write" locks are exclusive, as they are compatible with no other mode. The usage of a domain allows a lock to be separated from the data being locked.

The support provided by Aeolus for programming with actions is rather low level. The language provides direct access to the Clouds primitives. Programmers may write their own *action event handlers*, procedures that are called when an action event (such as commit or abort) happens.

Aeolus gives the programmer more flexibility than Argus for optimizing recovery and synchronization. On the negative side, the many features thus introduced make Aeolus a fairly complex language.

(4) *Implementation and experience.* A compiler and run-time system for Aeolus

have been implemented. Aeolus has not yet been used for any major distributed applications.

3.2 Languages with Logically Shared Address Spaces

We now turn our attention from languages with logically distributed address spaces to languages providing logically shared address spaces. In particular, we look at three subclasses: parallel functional languages, parallel logic languages, and languages based on distributed data structures (see Figure 4). Languages based on shared variables (e.g., Algol 68 [van Wijngaarden et al. 1975], Concurrent Pascal [Brinch Hansen 1975], and MESA [Geschke et al. 1977]) are not discussed here. They can (at least in principle) be implemented on a distributed system, using techniques like Shared Virtual Memory [Li and Hudak 1986], but they were designed for shared-memory multiprocessors. (For a detailed discussion of shared-variable languages, we refer the reader to Andrews and Schneider [1983].)

3.2.1 Parallel Functional Languages

Pure functional languages are being studied in several parallel programming projects (see Table 10). The implicit parallelism in functional languages is especially suited for closely coupled architectures like dataflow machines; whereas distributed computing systems are in general more coarse grained. Nevertheless, functional languages can be used for programming distributed systems by providing a mapping notation that efficiently distributes computations among processors. This approach is taken by the language ParAlfl (discussed below).

Nonpure functional languages can also be based on functional parallelism, but they require a mechanism for determining which expressions can be evaluated in parallel. The language FX-87 uses an *effect system* for this purpose. An effect is a static description of the side effects of an expression. The effect of a function can be specified by the programmer and checked by the compiler. The compiler uses the effect information to do certain optimizations and to determine which expressions to evaluate in parallel.

Table 10. Parallel Functional Languages

Parallel functional languages	
Language	References
Blaze	[Mehrotra and van Rosendale 1987]
Concurrent LISP	[Sugimoto et al. 1983]
FX-87	[Jouvelot and Gifford 1988]
Lisptalk	[Li 1988]
Multilisp	[Halstead 1985]
ParAlfl	[Hudak 1986]
PML	[Reppy 1988]
QLISP	[Gabriel and McCarthy 1984, 1988]
Symmetric LISP	[Gelernter et al. 1987a, 1987b]

Multilisp, QLISP, and Concurrent LISP are intended primarily for shared-memory machines and would be less efficient on distributed systems. Blaze is a Pascal-based language for parallel scientific programming that supports the functional programming model. It uses functional parallelism as well as explicit parallelism through a parallel loop-construct.

ParAlfl. ParAlfl [Hudak and Smith 1986; Hudak 1986, 1988] is a parallel functional language developed by Paul Hudak at Yale University.

(1) *Parallelism.* ParAlfl employs implicit, functional parallelism. Functional parallelism is usually fine grained, resulting in many small tasks that can be done in parallel. As there may be far more parallel tasks than physical processors, ParAlfl uses a mapping notation to specify which expressions are to be evaluated on which processors. An expression followed by the annotation **$on proc** will be evaluated on the processor determined by the expression **proc**. This **proc** expression can be relative to the currently executing processor. For example, if the expression

$$(f(x) \ \$\mathrm{on}\ (\$\mathrm{self} - 1)) \\ + (g(y) \ \$\mathrm{on}\ (\$\mathrm{self} + 1))$$

is executed by Processor P, then Processor $P - 1$ executes $f(x)$, Processor $P + 1$

executes $g(y)$ (in parallel), and Processor P itself performs the addition.

(2) *Communication and synchronization.* Communication and synchronization between parallel computations are implicit, so there is no need for explicit language constructs. A computation automatically blocks when it needs the result of another computation that is not yet available.

The semantics of ParAlfl are based on *lazy evaluation*, which means that an expression is not evaluated until its result is needed. In general, the programmer need not be concerned with the order in which computations are done. For efficiency reasons, he or she may want to control the evaluation order, however. For this purpose, ParAlfl supports *eager expressions*, which are evaluated before their results are needed, and *synchronizing expressions*, which constrain the evaluation order.

ParAlfl programs are fully deterministic, provided that a few simple restrictions on **proc** expressions are satisfied. This means that the results of such programs do not depend on how the computations are distributed among the processors. In particular, the results of a program will be the same whether executed on a uniprocessor or on a parallel system.

(3) *Implementation and experience.* ParAlfl has been implemented on the Encore Multimax multiprocessor and on two distributed architectures (hypercubes): the Intel iPSC and the NCube [Goldberg and Hudak 1986]. The language has been used for implementing several parallel algorithms (e.g., divide-and-conquer, linear equations, partial differential equations) [Hudak 1986].

3.2.2 Parallel Logic Languages

Many of the underlying ideas of parallel[14] logic programming languages (see Table 11) were introduced by Clark and Gregory for their Relational Language. Most parallel logic languages are based on AND/OR parallelism, shared logical variables, and committed choice nondeterminism (see Sections 2.1.1, 2.2.2, and 2.2.3). Examples are Concurrent PROLOG and Flat Concurrent PROLOG (discussed below), PARLOG (also discussed below), guarded Horn clauses (GHC), and Oc. P-PROLOG is also based on shared logical variables, but uses a mechanism called *exclusive guarded Horn clauses* for controlling OR-parallelism. For a normal guarded Horn clause, if several clauses for a given goal have a guard that evaluates to "true," then one of them is chosen nondeterministically. For an *exclusive* guarded Horn clause, however, the execution of the goal *suspends*, until exactly one guard evaluates to "true." (Note that a guard that initially succeeds can later fail, if one of the variables used by the guard gets bound.)

Table 11. Parallel Logic Languages

Parallel logic languages	
Language	References
BRAVE	[Reynolds et al. 1988]
Concurrent PROLOG	[Shapiro 1987]
Delta PROLOG	[Pereira et al. 1986]
Guarded Horn clauses	[Ueda 1985]
Mandala	[Ohki et al. 1987]
Oc	[Takeuchi and Furukawa 1986]
PARLOG	[Clark and Gregory 1986]
P-PROLOG	[Yang and Aiso 1986; Yang 1988]
Quty	[Sato 1987]
Relational Language	[Clark and Gregory 1981]
Vulcan	[Kahn et al. 1986]

BRAVE is a parallel logic language that does not use committed choice nondeterminism, but supports true OR-parallelism. Mandala combines object-oriented and logic programming. Quty combines functional and logic programming.

Delta PROLOG is significantly different from the languages mentioned above. It is based on message passing rather than on shared logical variables, it uses only AND-parallelism, and it supports PROLOG's completeness of search by using distributed backtracking.

[14] The logic programming community has adopted the term *concurrent* logic language rather than *parallel* logic language; for consistency with the rest of the paper, we use the latter term, however.

Concurrent PROLOG. Concurrent PROLOG was designed by Ehud Shapiro of the Weizmann Institute of Science in Rehovot, Israel [Shapiro 1983, 1986, 1987]. Concurrent PROLOG uses many of the ideas proposed by Clark and Gregory for their Relational Language. Shapiro and his group, however, have developed several new programming techniques for languages like Concurrent PROLOG.

(1) *Parallelism.* Parallelism in Concurrent PROLOG comes from the AND-parallel evaluation of the goals of a conjunction and from the OR-parallel evaluation of the guards of a guarded Horn clause, as discussed in Sections 2.1.1 and 2.2.3. There is no sequential AND-operator, so every goal of a conjunction creates a new parallel process. The textual ordering of the goals has no semantic significance. A mapping notation has been designed for assigning processes to processors, as discussed in Section 2.1.2.

(2) *Communication and synchronization.* Parallel processes communicate through shared logical variables. Synchronization is based on suspension on *read-only variables.* A variable is marked as read-only by suffixing it with a "?." Unification of two terms suspends if an attempt is made to instantiate a read-only variable. Thus, Concurrent PROLOG extends the unification algorithm of PROLOG [Robinson 1965] with a test for read-only variables.

Concurrent PROLOG uses guarded Horn clauses to deal with nondeterminism. There is no restriction on the kinds of goals that may appear in a guard, so a guard may create other AND-parallel processes. As these processes may invoke new guards, this may lead to a system of arbitrarily nested guards. This creates a problem, as only the guard of the clause that is committed to may have side effects (see Section 2.2.3). Therefore, a new *environment* is created for every guard of a guarded Horn clause, containing the bindings made by that guard. On commitment, the environment of the chosen guard is unified with the goal being solved. The environments of all other guards are discarded. Maintenance of these separate environments is difficult to implement, even on a single-processor machine. The need for environments has been eliminated in a subsequent language, called Flat Concurrent PROLOG (FCP) [Mierowsky et al. 1985]. In FCP, guards may only contain a predefined set of predicates, rather than user-defined predicates, so nesting of guards is ruled out. This also virtually eliminates OR-parallelism, but a method has been designed to compile OR-parallel programs into AND-parallel programs [Codish and Shapiro 1986].

(3) *Implementation and experience.* A uniprocessor implementation of Flat Concurrent PROLOG exists for several types of UNIX machines [Houri and Shapiro 1986]. The implementation supports the Logix programming environment and operating system [Silverman et al. 1986]. The novelty in the implementation is its efficient support for the creation, suspension, activation, and termination of lightweight processes. The performance is comparable to conventional uniprocessor PROLOG implementations.

A distributed implementation of FCP was developed for the Intel iPSC hypercube [Taylor et al. 1987b]. The key concepts in the implementation are data distribution by demand-driven structure copying and the use of a specialized two-phase locking protocol to implement FCP's atomic unification.

Several applications have been written in Concurrent PROLOG. Shapiro [1987] contains separate papers on Concurrent PROLOG implementations of systolic algorithms, the Maxflow problem (determining the maximum flow through a network), region finding in a self intersecting polygon, image processing, the Logix system, a distributed window system, a public-key system, an equation solver, a compiler for FCP, and a hardware simulator. Most experiences reported are quite positive; actual performance measurements are absent in nearly all papers.

Several programming techniques have been developed that can be used for systems and application programming in

Concurrent PROLOG. Streams, bounded buffers, and incomplete messages were mentioned in Section 2.2.2. Streams and merging of streams can be expressed in Concurrent PROLOG [Shapiro and Mierowsky 1984; Shapiro and Safra 1986]. The "short-circuit" technique implements distributed termination detection. "Metaprogramming" and "metainterpreters" are studied by Safra and Shapiro [1986]. Systolic programming is a well-known technique for executing numerical algorithms in parallel on special-purpose hardware [Kung 1982]. Shapiro [1984] shows that systolic algorithms can also be expressed in Concurrent PROLOG, so they can be run on general-purpose hardware. Concurrent PROLOG can also be used for object-oriented programming [Shapiro and Takeuchi 1983]. Kahn et al. have designed a preprocessor language for Concurrent PROLOG (called Vulcan), which allows object-oriented programs to be written with less verbosity [Kahn et al. 1986].

PARLOG. PARLOG is a parallel logic programming language being developed at Imperial College, London, by Keith Clark and Steve Gregory [Clark and Gregory 1985, 1986; Foster et al. 1986; Gregory 1987; Ringwood 1988; Clark 1988]. It is a descendant of IC-PROLOG [Clark et al. 1982] and the Relational Language [Clark and Gregory 1981]. Like Concurrent PROLOG, PARLOG is based on AND/OR parallelism and committed choice nondeterminism. The main innovation introduced by the language is the use of *mode declarations* to control synchronization.

(1) *Parallelism.* PARLOG uses AND/OR parallelism that can be controlled by the programmer. There are two different conjunction operators: "," evaluates both conjuncts in parallel, and "&" evaluates them sequentially (left to right). The clauses for a relation can be separated either by a "." or by a ";" operator. In finding a matching clause for a goal, all clauses separated by a "." are tried in parallel (OR-parallelism). Clauses after a ";" are only tried if all clauses before the separator do not match. In the example below,

1. $A \leftarrow (B \ \& \ C), (D \ \& \ E)$;
2. $A \leftarrow F, G$.
3. $A \leftarrow H \ \& \ J$.

Clause 1 is tried first, by doing "$(B \ \& \ C)$" and "$(D \ \& \ E)$" in parallel. If Clause 1 fails, Clauses 2 and 3 are tried in parallel. F and G are evaluated in parallel, but H and J are done sequentially.

The presence of sequential AND/OR operators requires the implementation to determine when a group of parallel processes has terminated, which is not a trivial task in a distributed environment. For example, in "B & C," all the processes created by B must have terminated before C is started. This additional complexity is the main reason why Concurrent PROLOG supports only the parallel operators.

(2) *Communication and synchronization.* Processes communicate through shared logical variables and synchronize by suspending on unbound shared variables. PARLOG has a mechanism for specifying which processes may generate a binding for a variable. For every relation, a *mode declaration* must be given that specifies which arguments are input and which are output. For example, the declaration

mode append(list1?, list2?,
appended-list^).

defines the first two arguments to the append relation to be input and the third one to be output. An actual argument appearing in an input position will only be used for input matching. If unification of the argument with the corresponding term in the head can only succeed by binding a variable appearing in the input argument, then the unification will *suspend.* The unification will be resumed when some *other* process generates a binding for the variable. After commitment, any actual argument appearing in an output position is unified with the output argument in the head of the clause.

PARLOG uses three specialized unification primitives for input matching, equality testing, and output unification. In contrast, Concurrent PROLOG has a general unification algorithm, which also has to take care of read-only variables.

Like Concurrent PROLOG, PARLOG uses guarded Horn clauses for nondeterminism. A guard in PARLOG may *test* any input variables and bind local variables of the clause, but it may not bind variables passed in an input argument. This is checked at compile time, using mode declarations. If a guard tries to bind an output variable, the actual binding is established only *after* commitment. Unlike Concurrent PROLOG, no environments need be maintained.

(3) *Implementation and experience.* The compiler can use the information in a mode declaration to increase efficiency. A sequential implementation of PARLOG is described by Foster et al. [1986]. The compiler first compiles PARLOG programs into a subset called Kernel PARLOG, in which all unifications are performed by explicit unification operators. Kernel PARLOG programs are subsequently compiled to code for an abstract machine, called the Sequential PARLOG Machine (SPM), which is emulated on a real machine.

A distributed implementation of PARLOG on a network of SUNs is described by Foster [1988]. The implementation uses some of the ideas of the distributed FCP implementation (see above), but differs in supporting distributed termination and deadlock detection. Also, as PARLOG does not have atomic unification, distributed unification is implemented without using a two-phase locking protocol.

PARLOG has been used for discrete event simulation, the specification and verification of communication protocols, a medical diagnosis expert system, and natural-language parsing [Gregory 1987; Clark 1988].

3.2.3 Distributed Data Structures

Distributed data structures are used in Linda, Orca (both discussed below), SDL, and Tuple Space Smalltalk (see Table 12).

Table 12. Languages Based on Distributed Data Structures

Distributed data structures	
Language	References
Linda	[Ahuja et al. 1986]
Orca	[Bal and Tanenbaum 1988]
SDL	[Roman et al. 1988]
Tuple Space Smalltalk	[Matsuoka and Kawai 1988]

Linda. Linda is being developed by David Gelernter and colleagues at Yale University [Gelernter 1985; Ahuja et al. 1986; Carriero et al. 1986]. Linda is not based on shared variables or message passing, but uses a novel communication mechanism: the Tuple Space. It supports (but does not enforce) a programming methodology based on distributed data structures and replicated workers. The goal of this methodology is to release the programmer from thinking in terms of parallel computations and simultaneous events, hence making parallel programming conceptually similar to sequential programming.

(1) *Parallelism.* Linda provides a simple primitive (called **eval**) to create a sequential process.[15] Linda does not have a notation for mapping processes to processors. With the replicated worker model (discussed below), there is no need for such a notation, as each processor executes a single process.

(2) *Communication and synchronization.* Linda's underlying communication model, the Tuple Space (TS), was discussed in Section 2.2.2. Processes communicate by inserting new tuples into TS and by reading or removing existing tuples. Processes synchronize by waiting for tuples to be available, using blocking **read** and **in** operations.

Traditional communication primitives (e.g., message passing and remote procedures) can be simulated using operations on TS [Gelernter 1985], so algorithms that split up the work among several communicating processes can be expressed in Linda.

[15] An earlier version of Linda provided constructs for parallel execution of a group of statements [Gelernter 1985]. We describe the current version here, which is based on C.

Alternatively, Linda programs can use the so-called *replicated workers* style [Ahuja et al. 1986]. Such a program consists of *P* identical (replicated) worker processes, one for each processor. The work to do is stored in a *distributed data structure*, which is implemented in TS and is accessible by all worker processes. Each process repeatedly takes some work from the data structure, performs it, puts back the results into the data structure, and possibly generates some more work. All workers essentially perform the same kind of task, until all work is done. The workers are loosely coupled; they only interact indirectly through the data structure. This model is claimed to have several advantages [Carriero et al. 1986]. In principle, any number of processors can be used (including just one). Also, the work is automatically and fairly distributed among the workers. Finally, process management is easy, as there usually is only one process per processor.

(3) *Fault tolerance.* A fault-tolerant network kernel for Linda, based on replication of the TS, has been designed by Xu [Xu 1988].

(4) *Implementation and experience.* Implementations exist for running Linda programs on Bell Labs' S/Net [Carriero and Gelernter 1986], an Ethernet-based MicroVax network, the iPSC hypercube [Gelernter and Carriero 1986], the Encore Multimax, the Sequent Balance, and other configurations. Different implementation strategies are discussed by Carriero [1987].

A hardware coprocessor has been designed by Venkatesh Krishnaswamy that supports Linda communication patterns and tuple matching [Ahuja et al. 1988]. Several (on the order of thousands) nodes, consisting of some CPU and the Linda coprocessor, can be arranged into a grid using this new hardware to form a highly parallel Linda Machine.

One of Linda's main goals is to achieve high speedups for real-life problems. Applications for which Linda programs have been written include DNA-sequence comparison, database search, VLSI simulation, heuristic monitoring, the Traveling Salesman problem, parameter sensitivity analysis, ray tracing, numerical problems [Gelernter and Carriero 1988], and a distributed backtracking package [Kaashoek and Bal 1988].

Orca. Orca is being developed by Henri Bal and Andrew Tanenbaum at the Vrije Universiteit in Amsterdam [Bal and Tanenbaum 1988; Bal 1989]. The language is primarily intended for the implementation of parallel algorithms on distributed systems. Orca allows processes on different processors to share data structures that are ençapsulated in passive objects, which are instances of abstract data types.

(1) *Parallelism.* Parallelism in Orca is based on sequential processes. Orca provides an explicit **fork** primitive for spawning a new child process and passing parameters to it. Parameters may be **value** or **shared**, as specified in the declaration of the child process.

With value parameters, a copy of the actual parameter is created and passed to the child process. This mode is allowed for any type of parameter, including data structures like sets and graphs. Unlike most procedural languages, Orca does not provide pointers for building graphs; instead, graphs are built-in data types, just like arrays and sets. This eliminates the problem described in Section 2.2.1 of passing complex data structures around in a distributed system.

The second parameter mode—shared—is only allowed for parameters of (user-defined) *abstract data types.* The actual parameter must be a variable of the same abstract type. The variable, called a *data object*, is shared by the parent and child process. A child process may pass the object as **shared** parameter to *its* children, and so on, so, in general, there will be a hierarchy of processes sharing objects.

(2) *Communication and synchronization.* Processes communicate indirectly through shared data objects. As described above, such objects are instances of abstract data types. An abstract data type definition consists of a **specification** part and an **implementation** part. The speci-

fication part lists the operations that can be applied to objects of the given type. Each operation is applied to a single object; other objects (or regular data) can be passed as value parameters. All operations to the same object are executed *indivisibly*. For example, if X is a shared object of type IntegerObject, and the specification part of IntegerObject contains the operation

operation increment(by: integer);

then the invocation

X\$increment(12);

applies the operation increment to object X, using the constant 12 as value parameter. If multiple processes simultaneously try to increment object X, then all these invocations will (at least conceptually) be serialized.

The **implementation** part of an abstract data type specifies the data contained in each variable (object) of the abstract type and contains code implementing the operations. An operation implementation may consist of one or more *guarded statements*. If so, an invocation of the operation blocks until at least one of the guards succeeds; next, one true guard is chosen nondeterministically, and its statements are executed without blocking again. An operation cannot block halfway during its execution.

(3) *Implementation and experience.* The shared data-object model can be implemented efficiently on a distributed system by *replicating* objects. If a shared object is changed infrequently, communication overhead is decreased by maintaining copies of the object on those processors that read it frequently and by applying read-only operations to the local copy. There are several different ways of deciding where to store copies of an object and how to update all these copies in a consistent way [Bal and Tanenbaum 1988]. One prototype implementation of Orca exists that replicates all objects on all processors and updates the copies using a reliable, ordered broadcast primitive (see Section 2.2.1). Orca has also been implemented on top of the Amoeba distributed operating system [Tanenbaum and van Renesse 1985]. This implementation uses selective replication of objects, based on statistics collected during execution by the run-time system.

A compiler front-end for Orca has been built using the Amsterdam Compiler Kit [Tanenbaum et al. 1983]. The compiler translates Orca programs into a machine-independent intermediate code, which is subsequently compiled into object code and linked with a machine-specific run-time system. The compiler also generates descriptive information to be used by the run-time system, such as which operations modify their objects and which are read-only.

Orca has been used for the implementation of several applications, including parallel branch-and-bound, computer chess, and graph algorithms.

4. CONCLUSIONS

The main reasons for running an application on a distributed computing system are high speed through parallelism, high reliability through replication of processes and data, and functional specialization. In addition, there are applications that are inherently distributed, such as sending electronic mail between geographically separated computers. The main issues distinguishing distributed programming from sequential programming are parallelism, communication and synchronization, and partial failures.

Parallelism was already employed in languages designed for implementing uniprocessor operating systems, which are easier to understand as collections of processes than as monolithic programs. The earliest languages were based on pseudoparallelism (no two processes are ever executed simultaneously), but with the advent of multiple-processor systems, interest shifted toward employing real parallelism to speed up programs. Parallelism was also welcomed by the designers of programming languages based on paradigms like logic programming, functional programming, and object-oriented programming. They realized that parallelism might be the solution to the problem of obtaining an efficient implementation of their languages. This has led to languages in which the details of managing parallelism are handled more by the run-time system and less by the programmer, yielding a higher level of abstraction.

The early pseudoparallel languages and operating systems used shared variables for interprocess communication. Mechanisms like semaphores and monitors were invented for synchronizing access to shared data in a clean way. A different approach was taken in the RC4000 operating system [Brinch Hansen 1973], which used message passing for interprocess communication. Later, message passing was also introduced as a programming language primitive [Hoare 1978]. This resulted in many Pascal-like languages based on some form of message passing. These languages were used for programming distributed systems as well as shared-memory multiprocessors. The next step was the development of higher level paradigms for interprocess communication, such as rendezvous, remote procedure calls (RPC), and distributed data structures.

A similar development took place in the area of techniques for fault-tolerant distributed applications. Early languages for distributed programming left the programmer to deal with fault tolerance. Later on, languages based on atomic transactions were introduced. The atomic transaction is a powerful mechanism for managing parallel access to data. Languages based on this abstraction take care of a lot of low-level details, such as locking and version management.

As a result of all these developments, we see several languages that differ significantly from their Pascal-based predecessors and that provide high-level abstractions for distributed programming. Unlike many of their predecessors, however, most novel languages have yet to establish themselves as practical tools for the development of distributed software. As the advances in hardware technology do not show any signs of slowing down, we expect distributed architectures to continue to become more widely available, more generally used, and to be programmed in increasingly advanced languages. A summary of the languages mentioned in this paper is given in Table 13.

APPENDIX

Table 13. Overview of Languages for Distributed Systems

Language	Section	Description
ABCL/1	3.1.6	Object-oriented language for modeling distributed systems
Act 1	3.1.6	Language based on actor model
Ada	3.1.3	Language based on rendezvous
Aeolus	3.1.7	Language based on atomic transactions, used for Clouds distributed operating system
ALPS	3.1.6	Object-oriented language for parallel and distributed systems
AMPL	3.1.2	Language based on asynchronous message passing
Argus	3.1.7	Language based on atomic transactions
Avalon	3.1.7	Language based on atomic transactions, used for Camelot distributed transaction system
Blaze	3.2.1	Language for scientific programming, based on parallel loops and functional (implicit) parallelism
BNR Pascal	3.1.3	Language based on rendezvous
BRAVE	3.2.2	Logic language for artificial-intelligence applications
Camelot Library	3.1.7	Language based on atomic transactions, used for Camelot distributed transaction system
Cantor	3.1.6	Language based on actor model
CCSP	3.1.1	CSP-based language for operating-system design
Cedar	3.1.4	Language based on remote procedure calls
CLIX	3.1.6	Object-oriented language
Cluster 86	3.1.6	Object-oriented language
CMAY	3.1.2	FORTRAN-based language with asynchronous message passing
Concurrent C	3.1.2	Language based on asynchronous message passing
Concurrent C	3.1.3	Extension of C with processes and rendezvous
Concurrent CLU	3.1.4	Language based on remote procedure calls
Concurrent LISP	3.2.1	Functional language with processes and shared variables
Concurrent PROLOG	3.2.2	Logic language based on read-only variables
ConcurrentSmalltalk	3.1.6	Object-oriented language
CONIC	3.1.2	Configuration language with asynchronous message passing
CSM	3.1.1	Language based on synchronous message passing
CSP-S	3.1.1	Language based on synchronous message passing
CSP/80	3.1.1	Language based on synchronous message passing
CSP	3.1.1	Language based on synchronous message passing
CSPS	3.1.1	CSP-based language for modeling distributed systems
CSSA	3.1.6	Pascal-based language using actor model, used for INCAS project
Delta PROLOG	3.2.2	Logic language based on message passing and distributed backtracking
Dislang	3.1.5	Language with multiple communication primitives
Distributed Smalltalk	3.1.6	Object-oriented language
DP	3.1.4	Language based on remote procedure calls
DPL-82	3.1.2	Language based on asynchronous message passing
ECSP	3.1.1	Language based on synchronous message passing
Emerald	3.1.6	Object-based language, employs object mobility
EPL	3.1.6	Object-based language, used for Eden distributed operating system
FRANK	3.1.2	Language based on asynchronous message passing
FX-87	3.2.1	Functional language based on effect system
GDPL	3.1.1	Language based on synchronous message passing
GHC	3.2.2	Logic language based on guarded Horn clauses
GYPSY	3.1.2	Language for implementing verifiable programs, based on asynchronous message passing
Hybrid	3.1.6	Object-oriented language
Joyce	3.1.1	Secure language based on CSP and Pascal
LADY	3.1.2	Language based on asynchronous message passing, used for INCAS project
LIMP	3.1.1	Language based on synchronous message passing
Linda	3.2.3	Language based on Tuple Space model
Lisptalk	3.2.1	Functional language based on CSP and LISP
LYNX	3.1.4	Language based on remote procedure calls
MC	3.1.3	Language based on rendezvous
Mandala	3.2.2	Logic/object-oriented language for knowledge programming
Mentat	3.1.6	Object-oriented language based on macro data flow
MENYMA/S	3.1.2	Language based on asynchronous message passing

Table 13. *(Continued)*

Language	Section	Description
Multilisp	3.2.1	Functional language
NIL	3.1.2	Secure language based on process model, typestates, and asynchronous message passing
Oc	3.2.2	Logic language
Occam	3.1.1	Language based on synchronous message passing, used for Inmos transputers
OIL	3.1.6	Object-oriented/logic/procedural language, used for FAIM-1 multiprocessor
Ondine	3.1.6	Object-oriented language
Orca	3.2.3	Language based on shared data objects
Orient84/K	3.1.6	Object-oriented language for modeling knowledge systems
P^+	3.1.4	Language based on remote procedure calls
P-PROLOG	3.2.2	Logic language based on exclusive guarded Horn clauses
ParAlfl	3.2.1	Functional language with mapping notation
PARLOG	3.2.2	Logic language based on mode declarations
ParMod	3.1.2	Pascal-based language with modules and asynchronous message passing
Pascal+CSP	3.1.1	Language integrating Pascal and CSP
Pascal-FC	3.1.5	Language with multiple communication primitives
Pascal-m	3.1.1	Language with synchronous message passing through mailboxes
PCL	3.1.2	Language based on asynchronous message passing
Planet	3.1.1	Pascal-based language with synchronous message passing through links
Platon	3.1.2	Pascal-based language with asynchronous message passing
PLITS	3.1.2	Language based on asynchronous message passing
PML	3.2.1	Functional language
POOL	3.1.6	Object-oriented language, used for Philips DOOM machine
Port Language	3.1.2	Language with asynchronous message passing through ports
Pronet	3.1.2	Language based on asynchronous message passing
Quty	3.2.2	Language combining logic and functional programming
QLISP	3.2.1	Functional language
Raddle	3.1.6	Object-based language for designing large distributed systems
RBCSP	3.1.1	Language based on synchronous message passing
Relational Language	3.2.2	Logic language, predecessor of PARLOG and Concurrent PROLOG
SDL	3.2.3	Language based on shared dataspace, extends tuple space with atomic transactions
SINA	3.1.6	Object-oriented language
Sloop	3.1.6	Object-oriented language based on virtual object space
SR	3.1.5	Language with multiple communication primitives
StarMod	3.1.5	Language with multiple communication primitives
Symmetric LISP	3.2.1	Parallel functional language
Vulcan	3.2.2	Logic object-oriented language
ZENO	3.1.2	Language based on asynchronous message passing

ACKNOWLEDGMENTS

We would like to thank Ehud Shapiro for his contribution to our discussion of parallel logic programming languages. Peter Wegner suggested many improvements to the paper, including the distinction between logical and physical distribution. We are grateful to Nick Carriero, David Cheriton, Per Brinch Hansen, and the anonymous referees, who gave us useful comments on various parts of the paper. Greg Andrews, Andrew Black, Narain Gehani, Steve Gregory, Tony Hoare, Paul Hudak, Norman Hutchinson, Gary Leavens, Barbara Liskov, and Tom Wilkes helped us in getting the description of their languages accurate and up-to-date. Erik Baalbergen, Susan Flynn Hummel, Dick Grune, Frans Kaashoek, Robert Halstead, Clifford Neuman, Robbert van Renesse, Guido van Rossum, Irvin Shizgal, Chris Steketee, Kuo-Chung Tai, and Hans Tebra were very kind to read and comment on earlier versions of the paper. Finally, we would like to thank the usenet mod.os readership for contributions to the "what is a distributed system" discussion.

REFERENCES

ABRAMSKY, S., AND BORNAT, R. 1983. Pascal-m: A language for loosely coupled distributed systems. In *Distributed Computing Systems*, Y. Paker and J.-P. Verjus, Eds. Academic Press, London, pp. 163–189.

Ackerman, W. B. 1982. Data flow languages. *Computer 15*, 2 (Feb.), 15–25.

Adamo, J.-M. 1982. Pascal+CSP, merging Pascal and CSP in a parallel processing oriented language. In *Proceedings of the 3rd International Conference on Distributed Computing Systems* (Miami/Ft. Lauderdale, Fla., Oct. 18–22). IEEE, New York, pp. 542–547.

Agha, G. 1986. An overview of actor languages. *SIGPLAN Not.* (ACM) *21*, 10 (Oct.), 58–67.

Ahuja, S., Carriero, N., and Gelernter, D. 1986. Linda and friends. *Computer 19*, 8 (Aug.), 26–34.

Ahuja, S., Carriero, N., Gelernter, D., and Krishnaswamy, V. 1988. Matching language and hardware for parallel computation in the Linda machine. *IEEE Trans. Comput. C-37*, 8 (Aug.), 921–929.

Aksit, M., and Tripathi, A. 1988. Data abstraction mechanisms in SINA/st. In Proceedings of Object-Oriented Programming Systems, Languages and Applications 1988. *SIGPLAN Not.* (ACM) *23*, 11 (Nov.), 267–275.

Almes, G. T. 1986. The impact of language and system on remote procedure call design. In *Proceedings of the 6th International Conference on Distributed Computing Systems* (Cambridge, Mass., May 19–23). IEEE, New York, pp. 414–421.

Almes, G. T., Black, A. P., Lazowska, E. D., and Noe, J. D. 1985. The Eden system: A technical review. *IEEE Trans. Softw. Eng. SE-11*, 1 (Jan.), 43–59.

Ambler, A. L., Good, D. I., Browne, J. C., Burger, W. F., Cohen, R. M., Hoch, C. G., and Wells, R. E. 1977. GYPSY: A language for specification and implementation of verifiable programs. *SIGPLAN Not.* (ACM) *12*, 3 (Mar.), 1–10.

America, P. 1987. POOL-T: A parallel object-oriented language. In *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, Eds. MIT Press, Cambridge, Mass., pp. 199–220.

Andrews, G. R. 1981. Synchronizing resources. *ACM Trans. Program. Lang. Syst. 3*, 4 (Oct.), 405–430.

Andrews, G. R., 1982. The distributed programming language SR—Mechanisms, design and implementation. *Softw. Prac. Exper. 12*, 8 (Aug.), 719–753.

Andrews, G. R., and Olsson, R. A. 1986. The evolution of the SR programming language. *Distrib. Comput. 1*, 3 (July), 133–149.

Andrews, G. R., and Schneider, F. B. 1983. Concepts and notations for concurrent programming. *ACM Comput. Surv. 15*, 1 (Mar.), 3–43.

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

Athas, W. C., and Seitz, C. L. 1988. Multicomputers: Message-passing concurrent computers. Computer 21, 8 (Aug.), 9–24.

Baalbergen, E. H. 1988. Design and implementation of parallel make. *Comput. Syst. 1*, 2 (Spring), 135–158.

Backus, J. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Commun. ACM 21*, 8 (Aug.), 613–641.

Bagrodia, R., and Chandy, K. M. 1985. A microkernel for distributed applications. In *Proceedings of the 5th International Conference on Distributed Computing Systems* (Denver, Colo., May 13–17). IEEE, New York, pp. 140–149.

Baiardi, F., Ricci, L., and Vanneschi, M. 1984. Static type checking of interprocess communication in ECSP. In Proceedings of the SIGPLAN 84 Symposium on Compiler Construction. *SIGPLAN Not.* (ACM) *19*, 6 (June), 290–299.

Bal, H. E. 1989. The shared data-object model as a paradigm for programming distributed systems. Ph.D. dissertation, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, the Netherlands.

Bal, H. E., and Tanenbaum, A. S. 1988. Distributed programming with shared data. In *Proceedings of the IEEE CS 1988 International Conference on Computer Languages* (Miami, Fla., Oct. 9–13). IEEE, New York, pp. 82–91.

Bal, H. E., van Renesse, R., and Tanenbaum, A. S. 1987. Implementing distributed algorithms using remote procedure calls. In *Proceedings of the AFIPS National Computer Conference* (Chicago, Ill., June 15–18). AFIPS Press, Reston, Va., pp. 499–506.

Ball, J. E., Williams, G. J., and Low, J. R. 1979. Preliminary ZENO language description. *SIGPLAN Not.* (ACM) *14*, 9 (Sept.), 17–34.

Bennett, J. K. 1987. The design and implementation of distributed Smalltalk. In Proceedings of Object-Oriented Programming Systems, Languages and Applications 1987. *SIGPLAN Not.* (ACM) 22, 12 (Dec.), 318–330.

Bernstein, A. J. 1980. Output guards and nondeterminism in "Communicating Sequential Processes". *ACM Trans. Program. Lang. Syst. 2*, 2 (Apr.), 234–238.

Birman, K. P., and Joseph, T. A. 1987. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst. 5*, 1 (Feb.), 47–76.

Birrell, A. D., and Nelson, B. J. 1984. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1 (Feb.), 39–59.

Black, A., Hutchinson, N., Jul, E., and Levy, H. 1986. Object structure in the Emerald system. In Proceedings of Object-Oriented Programming Systems, Languages and Applications 1986. *SIGPLAN Not.* (ACM) *21*, 11 (Nov.), 78–86.

Black, A. P., Hutchinson, N. C., McCord, B. C., and Raj, R. K. 1984. *EPL Programmer's Guide.* Univ. of Washington, Seattle, June.

Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L. 1987. Distribution and abstract types in Emerald. *IEEE Trans. Softw. Eng. SE-13*, 1 (Jan.), 65–76.

BLOCH, J. J. 1988. The Camelot library. In *Guide to the Camelot Distributed Transaction Facility: Release 1*, A. Z. Spector and K. R. Swedlow, Eds. Carnegie-Mellon University, Pittsburgh, Pa., pp. 29–62.

BORG, A., BAUMBACK, J., AND GLAZER, S. 1983. A message system supporting fault tolerance. In *Proceedings of the 9th Symposium on Operating Systems Principles* (Bretton Woods, N.H., Oct. 10–13). ACM-SIGOPS, New York, pp. 90–99.

BRINCH HANSEN, P. 1973. *Operating System Principles.* Prentice-Hall, Englewood Cliffs, N.J.

BRINCH HANSEN, P. 1975. The programming language concurrent pascal. *IEEE Trans. Softw. Eng. SE-1*, 2 (June), 199–207.

BRINCH, HANSEN, P. 1978. Distributed processes: A concurrent programming concept. *Commun. ACM 21*, 11 (Nov.), 934–941.

BRINCH HANSEN, P. 1987. Joyce—A programming language for distributed systems. *Softw. Pract. Exper. 17*, 1 (Jan.), 29–50.

BURNS, A. 1988. *Programming in Occam 2.* Addison-Wesley, Wokingham, England.

BURNS, A., AND DAVIES, G. 1988. Pascal-FC: A language for teaching concurrent programming. *SIGPLAN Not.* (ACM) *23*, 1 (Jan.), 58–66.

BURNS, A., LISTER, A. M., AND WELLING, A. J. 1987. A review of Ada tasking. Lecture Notes in Computer Science, Vol. 262. Springer-Verlag, Berlin.

BURTON, F. W. 1984. Annotations to control parallelism and reduction order in the distributed evaluation of functional programs. *ACM Trans. Program. Lang. Syst. 6*, 2 (Apr.), 159–174.

CARPENTER, B. E., AND CAILLIAU, R. 1984. Experience with remote procedure calls in a real-time control system. *Softw. Pract. Exper. 14*, 9 (Sept.), 901–907.

CARRIERO, N. 1987. The implementation of tuple space machines. Res. Rep. 567, Ph.D. dissertation, Dept. of Computer Science, Yale Univ., New Haven, Conn., Dec.

CARRIERO, N., AND GELERNTER, D. 1986. The S/Net's Linda kernel. *ACM Trans. Comput. Syst. 4*, 2 (May), 110–129.

CARRIERO, N., GELERNTER, D., AND LEICHTER, J. 1986. Distributed data structures in Linda. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages* (St. Petersburg, Fla., Jan. 13–15). ACM, New York, pp. 236–242.

CLARK, K. L. 1988. PARLOG and its applications. *IEEE Trans. Softw. Eng. SE-14*, 12, (Dec.), 1792–1804.

CLARK, K. L., AND GREGORY, S. 1981. A relational language for parallel programming. In *Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture* (Portsmouth, N.H., Oct.). ACM, New York, pp. 171–178.

CLARK, K. L., AND GREGORY, S. 1985. Notes on the implementation of PARLOG. *J. Logic Program. 2*, 1 (Apr.), 17–42.

CLARK, K. L., AND GREGORY, S. 1986. PARLOG: Parallel programming in logic. *ACM Trans. Program. Lang. Syst. 8*, 1 (Jan.), 1–49.

CLARK, K. L., MCCABE, F. G., AND GREGORY, S. 1982. IC-PROLOG language features. In *Logic Programming*, S.-A. Tarnlund, Ed. Academic Press, London, pp. 253–266.

CMELIK, R. F., 1986. *Concurrent Make: The Design and Implementation of a Distributed Program in Concurrent C.* AT&T Bell Laboratories, Murray Hill, N.J.

CMELIK, R. F., GEHANI, N. H., AND ROOME, W. D. 1986. *Experience with Distributed Versions of Concurrent C.* AT&T Bell Laboratories, Murray Hill, N.J. (Also to appear in *IEEE Trans. Softw. Eng.*)

CMELIK, R. F., GEHANI, N. H., AND ROOME, W. D. 1987. *Fault Tolerant Concurrent C: A Tool for Writing Fault Tolerant Distributed Programs.* AT&T Bell Laboratories, Murray Hill, N.J.

CODISH, M., AND SHAPIRO, E. 1986. Compiling OR-parallelism into AND-parallelism. In *Proceedings of the 3rd International Conference on Logic Programming* (London, July 14–18), Springer-Verlag, Berlin, pp. 283–297.

COOK, R. P. 1980. *MOD—A language for distributed programming. *IEEE Trans. Softw. Eng. SE-6*, 6 (Nov.), 563–571.

COOPER, E. C. 1985. Replicated distributed programs. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Rosario Resort Orcas Island, Wash., Dec.). New York, ACM–SIGOPS, pp. 63–78.

COOPER, R. C. B., AND HAMILTON, K. G. 1988. Preserving abstraction in concurrent programming. *IEEE Trans. Softw. Eng. SE-14*, 2 (Feb.), 258–263.

COX, I. J., AND GEHANI, N. H. 1986. Concurrent programming and robotics. AT&T Bell Laboratories, Murray Hill, N.J.

CROOKES, D., AND ELDER, J. W. G. 1984. An experiment in language design for distributed systems. *Softw. Pract. Exper. 14*, 10 (Oct.), 957–971.

DANNENBERG, R. B. 1981. *AMPL: Design, implementation and evaluation of a multiprocessing language.* Carnegie-Mellon University. Pittsburgh, Pa.

DAVIS, A. L., AND ROBISON, S. V. 1985. The architecture of the FAIM-1 symbolic multiprocessing system. In *9th International Joint Conference on Artificial Intelligence* (Los Angeles, Calif., Aug. 13–18). pp. 32–38.

DAY, M. S. 1987. Replication and reconfiguration in a distributed mail repository. TR-376, MIT Laboratory for Computer Science, Cambridge, Mass., Apr.

DEPARTMENT OF DEFENSE, U.S. 1983. Reference manual for the Ada programming language. ANSI/MIL-STD-1815A, DoD,Washington, D.C., Jan.

DETLEFS, D. L., HERLIHY, M. P., AND WING, J. M. 1988. Inheritance of synchronization and recovery properties in Avalon/C++. *Computer 21*, 12 (Dec.), 57–69.

DIJKSTRA, E. W. 1975. Guarded commands, nondeterminacy, and formal derivation of programs. *Commun. ACM 18*, 8 (Aug.), 453–457.

EICHHOLZ, S. 1987. Parallel programming with ParMod. In *Proceedings of the 1987 International Conference on Parallel Processing* (St. Charles, Ill., Aug. 17–21). Penn State University, pp. 377–380.

ELRAD, T., AND MAYMIR-DUCHARME, F. 1986. Distributed languages design: Constructs for controlling preferences. In *Proceedings of the 1986 International Conference on Parallel Processing* (St. Charles, Ill., Aug. 19–22). Penn State University, pp. 176–183.

ERICSON, L. W. 1982. DPL-82: A language for distributed processing. In *Proceedings of the 3rd International Conference on Distributed Computing Systems* (Miami/Ft. Lauderdale, Fla., Oct.). pp. 526–531.

FELDMAN, J. A. 1979. High level programming for distributed computing. *Commun. ACM 22*, 6 (June), 353–368.

FINKEL, R., AND MANBER, U. 1987. DIB—A distributed implementation of backtracking. *ACM Trans. Program. Lang. Syst. 9*, 2 (Apr.), 235–256.

FISHER, A. J. 1986. A multi-processor implementation of Occam. *Softw. Pract. Exper. 16*, 10 (Oct.), 875–892.

FORMAN, I. R. 1986. On the design of large distributed systems. In *Proceedings of the IEEE CS 1986 International Conference on Computer Languages* (Miami, Fla., Oct. 27–30). IEEE, New York, pp. 84–95.

FOSTER, I. 1988. Parallel implementation of PARLOG. In *Proceedings of the International Conference on Parallel Processing* (*Vol. II*) (St. Charles, Ill., Aug. 15–19). Penn State University, pp. 9–16.

FOSTER, I., GREGORY, S., RINGWOOD, G., AND SATOH, K. 1986. A sequential implementation of PARLOG. In *Proceedings of the 3rd International Conference on Logic Programming* (London, July 14–18). Springer-Verlag, Berlin, pp. 149–156.

GABRIEL, R. P., AND MCCARTHY, J. 1984. Queue-based multi-processing Lisp. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming* (Austin, Tex., Aug. 6–8). ACM, New York, pp. 25–43.

GABRIEL, R. P., AND MCCARTHY, J. 1988. QLISP. In *Parallel Computation and Computers for Artificial Intelligence*, J. Kowalik, Ed. Kluwer Academic Publishers, Deventer, The Netherlands, pp. 63–89.

GAMMAGE, N. D., KAMEL, R. F., AND CASEY, L. M. 1987. Remote rendezvous. *Softw. Pract. Exper. 17*, 10 (Oct.), 741–755.

GEHANI, N. H. 1984a. *Ada: Concurrent Programming*. Prentice-Hall, Englewood Cliffs, N.J.

GEHANI, N. H. 1984b. Broadcasting sequential processes (BSP). *IEEE Trans. Softw. Eng. SE-10*, 4 (July), 343–351.

GEHANI, N. H. 1987. *Message Passing: Synchronous versus Asynchronous*. AT&T Bell Laboratories, Murray Hill, N.J.

GEHANI, N. H., AND CARGILL, T. A. 1984. Concurrent programming in the Ada language: The polling bias. *Softw. Pract. Exper. 14*, 5 (May), 413–427.

GEHANI, N. H., AND ROOME, W. D. 1986a. Concurrent C. *Softw. Pract. Exper. 16*, 9 (Sept.), 821–844.

GEHANI, N. H., AND ROOME, W. D. 1986b. *Concurrent C++: Concurrent Programming with Class(es)*. AT&T Bell Laboratories, Murray Hill, N.J.

GEHANI, N. H., AND ROOME, W. D. 1988. Rendezvous facilities: Concurrent C and the Ada language. *IEEE Trans. Softw. Eng. SE-14*, 11 (Nov.), 1546–1553.

GEHANI, N. H., AND ROOME, W. D. 1989. *The Concurrent C Programming Language*. Silicon Press, Summit, N.J.

GELERNTER, D. 1984. A note on systems programming in Concurrent Prolog. In *Proceedings of the International Symposium on Logic Programming* (Atlantic City, N.J., Feb. 6–9). IEEE, New York, pp. 76–82.

GELERNTER, D. 1985. Generative communication in Linda. *ACM Trans. Program. Lang. Syst. 7*, 1 (Jan.), 80–112.

GELERNTER, D., AND CARRIERO, N. 1986. Linda on hypercube multicomputers. In *Proceedings of the 1985 SIAM Conference* (Knoxville, Tenn.). Society for Industrial and Applied Mathematics, Philadelphia, Pa., pp. 45–55.

GELERNTER, D., AND CARRIERO, N. 1988. Applications experience with Linda. In *Proceedings of* PPEALS *1988. SIGPLAN Not.* (ACM) *23*, 9 (Sept.), 173–187.

GELERNTER, D., JAGANNATHAN, S., AND LONDON, T. 1987a. Environments as first class objects. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages* (Munich, West Germany, Jan. 21–23). ACM, New York.

GELERNTER, D., JAGANNATHAN, S., AND LONDON, T. 1987b. Parallelism, persistence and meta-cleanliness in the symmetric Lisp interpreter. In Proceedings of the Symposium on Interpreters and Interpretive Techniques. *SIGPLAN Not.* (ACM) *22*, 7 (July), 274–282.

GESCHKE, C. M., JR., MORRIS, J. H., AND SATTERTHWAITE, E. H. 1977. Early experience with Mesa. *Commun. ACM 20*, 8 (Aug.), 540–553.

GOLDBERG, A., AND ROBSON, D. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, Mass.

GOLDBERG, B., AND HUDAK, P. 1986. Alfalfa: Distributed graph reduction on a hypercube multiprocessor. Lecture Notes in Computer Science, vol. 279 (*Proceedings of the Santa Fe Graph Reduction Workshop*). Springer-Verlag, New York, pp. 94–113.

GRAHAM, P. C. J. 1985. Using BINS for interprocess communication. *SIGPLAN Not.* (ACM) *20*, 2 (Feb.), 32–41.

GREGORY, S. 1987. *Parallel Logic Programming in PARLOG.* Addison-Wesley, Wokingham, England.

GREIF, I., SELIGER, R., AND WEIHL, W. 1986. Atomic data abstractions in a distributed collaborative editing system. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages* (St. Petersburg, Fla., Jan.). ACM, New York, pp. 160–172.

GRIMSHAW, A. S., AND LIU, J. W. S. 1987. Mentat: An object-oriented macro data flow system. In Proceedings of Object-Oriented Programming Systems, Languages and Applications 1987. *SIGPLAN Not.* (ACM) *22*, 12 (Dec.), 35–47.

HALSTEAD, R. H., JR. 1985. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7*, 4 (Oct.), 501–538.

HAMILTON, K. G. 1984. A remote procedure call system. Tech. Rep. 70, Ph.D. dissertation, Computer Laboratory, Univ. of Cambridge, Cambridge, U.K., Dec.

HERLIHY, M., AND LISKOV, B. 1982. A value transmission method for abstract data types. *ACM Trans. Program. Lang. Syst. 4*, 4 (Oct.), 527–551.

HEWITT, C. 1977. Viewing control structures as patterns of passing messages. *Artif. Intell. 8*, 3 (June), 323–364.

HOARE, C. A. R. 1978. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug.), 666–677.

HOARE, C. A. R. 1981. The emperor's old clothes. *Commun. ACM 24*, 2 (Feb.), 75–83.

HOARE, C. A. R. 1985. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, N.J.

HOLT, R. C. 1982. A short introduction to Concurrent Euclid. *SIGPLAN Not.* (ACM) *17*, 5 (May), 60–79.

HOURI, A., AND SHAPIRO, E. 1986. A sequential abstract machine for Flat Concurrent Prolog. Rep. CS86-20, Dept. of Computer Science, The Weizmann Institute of Science, Rehovot, Israel, July.

HUDAK, P. 1986. Para-functional programming. *Computer 19*, 8 (Aug.), 60–70.

HUDAK, P. 1988. Exploring parafunctional programming: Separating the what from the how. *IEEE Softw. 5*, 1 (Jan.), 54–61.

HUDAK, P., AND SMITH, L. 1986. Para-functional programming: A paradigm for programming multiprocessor systems. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages* (St. Petersburg, Fla., Jan. 13–15). ACM, New York, pp. 243–254.

HULL, M. E. C., AND DONNAN, G. 1986. Contextually communicating sequential processes—a software engineering approach. *Softw. Pract. Exper. 16*, 9 (Sept.), 845–864.

HUNT, J. G., 1979. Messages in typed languages. *SIGPLAN Not.* (ACM) *14*, 1 (Jan.), 27–45.

HUR, J. H., AND CHON, K. 1987. Overview of a parallel object-oriented language CLIX. Lecture Notes in Computer Science, Vol. 276 (*Proceedings of the European Conference on Object-Oriented Programming*). Springer-Verlag, Berlin, pp. 265–273.

HUTCHINSON, N. C. 1987. Emerald: An object-based language for distributed programming. Tech. Rep. 87-01-01, Ph.D. dissertation, Dept. of Computer Science, Univ. of Washington, Seattle, Jan.

INMOS LTD. 1984. *Occam Programming Manual.* Prentice-Hall, Englewood Cliffs, N.J.

ISHIKAWA, Y., AND TOKORO, M. 1987. Orient84/K: An object-oriented concurrent programming language for knowledge representation. In *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, Eds. MIT Press, Cambridge, Mass., pp. 159–198.

JAZAYERI, M., GHEZZI, C., HOFFMAN, D., MIDDLETON, D., AND SMOTHERMAN, M. 1980. CSP/80: A language for communicating sequential processes. In *Proceedings of IEEE COMPCON Fall 1980* (New York, Sept.). IEEE, New York, pp. 736–740.

JONES, A. K., AND SCHWARZ, P. 1980. Experience using multiprocessor systems—A status report. *ACM Comput. Surv. 12*, 2 (June), 121–165.

JOSEPH, T. A., AND BIRMAN, K. P. 1986. Low cost management of replicated data in fault-tolerant distributed systems. *ACM Trans. Comput. Syst. 4*, 1 (Feb.), 54–70.

JOUVELOT, P., AND GIFFORD, D. K. 1988. The FX-87 interpreter. In *Proceedings of the IEEE CS 1988 International Conference on Computer Languages* (Miami, Fla., Oct. 9–13). IEEE, New York, pp. 65–72.

JUL, E. 1988. Object mobility in a distributed object-oriented system. Tech. Rep. 88-12-06, Ph.D. dissertation, Univ. of Washington, Seattle, Dec.

JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. 1988. Fine-grained mobility in the Emerald system. *ACM Trans. Comput. Syst. 6*, 1 (Feb.), 109–133.

KAASHOEK, M. F., AND BAL, H. E. 1988. An evaluation of the distributed data structure paradigm in Linda. IR-173, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, Dec.

KAHN, K., TRIBBLE, E. D., MILLER, M. S., AND BOBROW, D. G. 1986. Objects in concurrent logic programming languages. Proceedings of Object-Oriented Programming Systems, Languages and Applications 1986. *SIGPLAN Not.* (ACM) *21*, 11 (Nov.), 242–257.

VAN KATWIJK, J. VAN 1987. The Ada compiler. Ph.D. dissertation, Dept. of Mathematics and Computer Science, Delft Univ. of Technology, Delft, The Netherlands, Sept.

KERNIGHAN, B. W., AND RITCHIE, D. M. 1978. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, N.J.

KERRIDGE, J., AND SIMPSON, D. 1986. Communicating parallel processes. *Softw. Pract. Exper. 16*, 1 (Jan.), 63–86.

KIEBURTZ, R. A., AND SILBERSCHATZ, A. 1979. Comments on "Communicating Sequential Processes." *ACM Trans. Program. Lang. Syst. 1*, 2 (Oct.), 218–225.

KOCH, A., AND MAIBAUM, T. S. E. 1982. A message oriented language for system applications. In *Proceedings of the 3rd International Conference on Distributed Computing Systems* (Fort Lauderdale, Fla., Oct. 18–22). IEEE, New York, pp. 824–832.

KRAMER, J., AND MAGEE, J. 1985. Dynamic configuration for distributed systems. *IEEE Trans. Softw. Eng. SE-11*, 4 (Apr.), 424–436.

KRUATRACHUE, B., AND LEWIS, T. 1988. Grain size determination for parallel processing. *IEEE Softw. 5*, 1 (Jan.), 23–32.

KUNG, H. T. 1982. Why systolic architectures? *Computer 15*, 1 (Jan.), 37–46.

LAMPSON, B. W. 1981. Atomic transactions. In *Distributed Systems—Architecture and Implementation*, B. W. Lampson, Ed. Springer-Verlag, New York, pp. 246–265.

LEBLANC, R. J., AND MACCABE, A. B. 1982. The design of a programming language based on connectivity networks. In *Proceedings of the 3rd International Conference on Distributed Computing Systems* (Fort Lauderdale, Fla., Oct. 18–22). IEEE, New York, pp. 532–541.

LEBLANC, R. J., AND WILKES, T. 1985. Systems programming with objects and actions. In *Proceedings of the 5th International Conference on Distributed Computing Systems* (Denver, Colo., May 13–17). IEEE, New York, pp. 132–139.

LEBLANC, T. J., AND COOK, P. An analysis of language models for high-performance communication in local-area networks. *SIGPLAN Not.* (ACM) *18*, 6 (June), 65–72.

LESSER, V., SERRAIN, D., AND BONAR, J. 1979. PCL—A process oriented job control language. In *Proceedings of the 1st International Conference on Distributed Computing Systems* (Huntsville, Ala., Oct. 1–5). IEEE, New York, pp. 315–329.

LI, C. 1988. Concurrent programming language Lisptalk. *SIGPLAN Not.* (ACM) *23*, 4 (Apr.), 71–80.

LI, C.-M., AND LIU, M. T. 1981. Dislang: A distributed programming language/system. In *Proceedings of the 2nd International Conference on Distributed Computing Systems* (Paris, France, Apr. 8–10). IEEE, New York, pp. 162–172.

LI, K., AND HUDAK, P. 1986. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing* (Calgary, Canada, Aug. 13–18). ACM, New York, pp. 229–239.

LIEBERMAN, H. 1987. Concurrent object-oriented programming in Act 1. In *Object-Oriented Concurrent Programming*. A. Yonezawa and M. Tokoro, Eds. MIT Press, Cambridge, Mass., pp. 9–36.

LISKOV, B. 1979. Primitives for distributed computing. In *Proceedings of the 7th Symposium on Operating Systems Principles* (Pacific Grove, Calif., Dec. 10–12). ACM-SIGOPS, New York, pp. 33–42.

LISKOV, B. 1982. On linguistic support for distributed programs. *IEEE Trans. Softw. Eng. SE-8*, 3 (May), 203–210.

LISKOV, B. 1984. Overview of the Argus language and system. Programming Methodology Group Memo 40, MIT Laboratory for Computer Science, Cambridge, Mass., Feb.

LISKOV, B. 1988. Distributed programming in Argus. *Commun. ACM 31*, 3 (Mar.), 300–312.

LISKOV, B., AND SCHEIFLER, R. 1983. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst. 5*, 3 (July), 381–404.

LISKOV, B., CURTIS, D., JOHNSON, P., AND SCHEIFLER, R. 1987. Implementation of Argus. In *Proceedings of the 11th Symposium on Operating Systems Principles* (Austin, Tex., Nov. 8–11). ACM-SIGOPS, New York, pp. 111–122.

LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. 1977. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (Aug.), 564–576.

LUCCO, S. E., 1987. Parallel programming in a virtual object space. In Proceedings of Object-Oriented Programming Systems, Languages and Applications 1987. *SIGPLAN Not.* (ACM) *22*, 12 (Dec.), 26–34.

LUJUN, S., AND ZHONGXIU, S. 1987. An object-oriented programming language for developing distributed software. *SIGPLAN Not.* (ACM) *22*, 8 (Aug.), 51–56.

MARSLAND, T. A., OLAFSSON, M., AND SCHAEFFER, J. 1986. Multiprocessor tree-search experiments. In *Advances in Computer Chess 4*, D. F. Beal, Ed. Pergamon Press, Oxford, pp. 37–51.

MATSUOKA, S., AND KAWAI, S. 1988. Using tuple space communication in distributed object-oriented languages. In Proceedings of Object-Oriented Programming Systems, Languages and Applications 1988. *SIGPLAN Not.* (ACM) *23*, 11 (Nov.), 276–284.

MAY, D. 1983. Occam. *SIGPLAN Not.* (ACM) *18*, 4 (Apr.), 69–79.

MAY, D., AND SHEPHERD, R. 1984. The transputer implementation of Occam. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1984* (Tokyo, Japan, Nov. 6–9), pp. 533–541.

MEHROTRA, P., AND ROSENDALE, J. VAN. 1985. The Blaze language: A parallel language for scientific programming. *Parallel Comput. 5*, 3 (Nov.), 339–361.

MIEROWSKY, C., TAYLOR, S., SHAPIRO, E., LEVY, J., AND SAFRA, M. 1985. The design and implementation of Flat Concurrent Prolog. Rep. CS85-09, Dept. of Computer Science, The Weizmann Institute of Science, Rehovot, Israel, July.

MILEWSKI, J. 1984. Loglan implementation of the AMPL message-passing system. *SIGPLAN Not.* (ACM) *19*, 9 (Sept.), 21–29.

MOSS, J. E. B. 1981. Nested transactions: An approach to reliable distributed computing. Tech. Rep. MIT/LCS/TR-260, Ph.D. dissertation, MIT Laboratory for Computer Science, Cambridge, Mass.

MUNDIE, D. A., AND FISHER, D. A. 1986. Parallel processing in Ada. *Computer 19*, 8 (Aug.), 20–25.

MYERS, W. 1987. Ada: First users—Pleased; Prospective users—Still hesitant. *Computer 20*, 3 (Mar.), 68–73.

NEHMER, J., HABAN, D., MATTERN, F., WYBRANIETZ, D., AND ROMBACH, H. D. 1987. Key concepts in the INCAS multicomputer project. *IEEE Trans. Softw. Eng. SE-13*, 8 (Aug.), 913–923.

NELSON, B. J. 1981. Remote procedure call. Rep. CMU-CS-81-119, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., May.

NG, K.-W., AND LI, W. 1984. GDPL—A generalized distributed programming language. In *Proceedings of the 4th International Conference on Distributed Computing Systems* (San Francisco, Calif., May 14–18). IEEE, New York, pp. 69–78.

NIERSTRASZ, O. M. 1987. Active objects in hybrid. In Proceedings of Object-Oriented Programming Systems, Languages and Applications 1987. *SIGPLAN Not.* (ACM) *22*, 12 (Dec.), 243–253.

OGIHARA, T., KAJIHARA, Y., NAGANO, S., AND ARISAWA, M. 1986. Concurrency introduction to an object-oriented language system ondine. In *3rd National Conference Record A-5-1.* Japan Society for Software Science and Technology, Japan.

OHKI, M., TAKEUCHI, A., AND FURUKAWA, K. 1987. An object-oriented programming language based on the parallel logic programming language KL1. In *Proceedings of the 4th International Conference on Logic Programming* (Melbourne, Australia (May 25–29). MIT Press, Cambridge, Mass., pp. 894–909.

PAPERT, S. 1981. *Mindstorms: Children, Computers and Powerful Ideas.* Basic Books, New York.

PATNIAK, L. M., AND BADRINATH, B. R. 1984. Implementation of CSP-S for description of distributed algorithms. *Comput. Lang. 9*, 3, 193–202.

PEREIRA, L. M., MONTEIRO, L., CUNHA, J., AND APARICIO, J. N. 1986. Delta Prolog: A distributed backtracking extension with events. In *Proceedings of the 3rd International Conference on Logic Programming* (London, July 14–19). Springer-Verlag, Berlin, pp. 69–83.

POWELL, M. L., AND PRESOTTO, D. L. 1983. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the 9th Symposium on Operating Systems Principles* (Bretton Woods, N.H., Oct.). ACM-SIGOPS, New York, pp. 100–109.

RANKA, S., WON, Y., AND SAHNI, S. 1988. Programming a hypercube multicomputer. *IEEE Softw. 5*, 5 (Sept.), 69–77.

REPPY, J. H. 1988. Synchronous operations as first-class values. In *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation* (Atlanta, Ga., June 22–24). ACM, New York, pp. 250–259.

REYNOLDS, T. J., BEAUMONT, A. J., CHENG, A. S. K., DELGADO-RANNAURO, S. A., AND SPACEK, L. A. 1988. BRAVE—A parallel logic language for artificial intelligence. *Future Generations Comput. Syst. 4*, 1 (Aug.), 69–75.

RINGWOOD, G. A. 1988. PARLOG86 and the dining logicians. *Commun. ACM 31*, 1 (Jan.), 10–25.

RIZK, A., AND HALSALL, F. 1987. Design and implementation of a C-based language for distributed real-time systems. *SIGPLAN Not.* (ACM) *22*, 6 (June), 83–96.

ROBINSON, J. A. 1965. A machine-oriented logic based on the resolution principle. *J. ACM 12*, 1 (Jan.), 23–41.

ROMAN, G.-C., CUNNINGHAM, H. C., AND EHLERS, M. E. 1988. A shared dataspace language supporting large-scale concurrency. In *Proceedings of the 8th International Conference on Distributed Computing Systems* (San Jose, Calif., June 13–17). IEEE, New York, pp. 265–272.

ROMAN, G.-C., EHLERS, M. E., CUNNINGHAM, H. C., AND LYKINS, R. H. 1987. Toward comprehensive specification of distributed systems. In *Proceedings of the 7th International Conference on Distributed Computing Systems* (Berlin, Sept. 21–25). IEEE, New York, pp. 282–289.

ROOME, W. D. 1986. Discrete event simulation in Concurrent C. AT&T Bell Laboratories, Murray Hill, N.J.

ROPER, T. J., AND BARTER, J. 1981. A communicating sequential process language and implementation. *Softw. Pract. Exper. 11*, 11 (Nov.), 1215–1234.

RUSSELL, R. M. 1978. The CRAY-1 computer system. *Commun. ACM 21*, 1 (Jan.), 63–72.

SAFRA, S., AND SHAPIRO, E. 1986. Meta interpreters for real. In *Proceedings of IFIP Congress '86* (Dublin, Ireland, Sept.). IFIP, pp. 271–278.

SATO, M. 1987. Quty: A concurrent language based on logic and function. In *Proceedings of the 4th International Conference on Logic Programming* (Melbourne, Australia, May 25–29). MIT Press, Cambridge, Mass., pp. 1034–1056.

SCOTT, M. L. 1985. Design and implementation of a distributed systems language. Tech. Rep. 596, Ph.D. dissertation, Computer Science Dept., Univ. of Wisconsin at Madison, May.

SCOTT, M. L. 1986. The interface between distributed operating system and high-level programming language. In *Proceedings of the 1986 International Conference on Parallel Processing* (St. Charles, Ill., Aug. 19–22). Penn State University, pp. 242–249.

SCOTT, M. L. 1987. Language support for loosely-coupled distributed programs. *IEEE Trans. Softw. Eng. SE-13*, 1 (Jan. 1987), 88–103.

SCOTT, M. L., AND COX, A. L. 1987. An empirical study of message-passing overhead. In *Proceedings of the 7th International Conference on Dis-*

tributed Computing Systems (Berlin, Sept. 21–25). IEEE, New York, pp. 536–543.

SEITZ, C. L. 1985. The cosmic cube. *Commun. ACM 28*, 1 (Jan.), 22–33.

SHAPIRO, E. 1983. A subset of Concurrent Prolog and its interpreter. ICOT TR-003, Institute for New Generation Computer Technology, Tokyo, Japan, Feb.

SHAPIRO, E. 1984. Systolic programming: A paradigm of parallel processing. In *Proceedings of International Conference on Fifth Generation Computer Systems 1984* (Tokyo, Japan, Nov. 6–9). pp. 458–471.

SHAPIRO, E. 1986. Concurrent Prolog: A progress report. *Computer 19*, 8 (Aug.), 44–58.

SHAPIRO, E. 1987. *Concurrent Prolog: Collected Papers.* MIT Press, Cambridge, Mass.

SHAPIRO, E., AND MIEROWSKY, C. 1984. Fair, biased, and self-balancing merge operators: Their specifications and implementation in Concurrent Prolog. *J. New Generation Comput. 2*, 3, 221–240.

SHAPIRO, E., AND SAFRA, S. 1986. Multiway merge with constant delay in Concurrent Prolog. *J. New Generation Comput. 4*, 3, 211–216.

SHAPIRO, E., AND TAKEUCHI, A. 1983. Object-oriented programming in Concurrent Prolog. *J. New Generation Comput. 1*, 1, 25–48.

SILBERSCHATZ, A. 1984. Cell: A distributed computing modularization concept. *IEEE Trans. Softw. Eng. SE-10*, 2 (Mar.), 178–185.

SILVERMAN, W., HIRSCH, M., HOURI, A., AND SHAPIRO, E. 1986. The Logix system user manual. Rep. CS86-21, Dept. of Computer Science, The Weizmann Institute of Science, Rehovot, Israel.

SLOMAN, M., AND KRAMER, J. 1987. *Distributed Systems and Computer Networks.* Prentice-Hall, Englewood Cliffs, N.J.

SMITH-THOMAS, B. 1986. Managing I/O in concurrent programming: The Concurrent C window manager. AT&T Bell Laboratories, Murray Hill, N.J.

SPECTOR, A. Z., BLOCH, J. J., DANIELS, D. S., DRAVES, R. P., DUCHAMP, D., EPPINGER, J. L., MENEES, S. G., AND THOMPSON, D. S. 1986. The Camelot project. Rep. CMU-CS-86-166, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., Nov.

STAUNSTRUP, J. 1982. Message passing communication versus procedure call communication. *Softw. Pract. Exper. 12*, 3, (March) 223–234.

STROM, R. E. 1986. A comparison of the object-oriented and process paradigms. *SIGPLAN Not.* (ACM) 21, 10 (Oct.), 88–97.

STROM, R. E., AND YEMINI, S. 1983. NIL: An integrated language and system for distributed programming. *SIGPLAN Not.* (ACM) *18*, 6 (June), 73–82.

STROM, R. E., AND YEMINI, S. 1984. The NIL distributed systems programming language: A status report. In *Proceedings of the NSF/SRC Seminar Semantics of Concurrency* (Pittsburgh, Pa., July 9–11). Springer-Verlag, New York, pp. 512–523.

STROM, R. E., AND YEMINI, S. 1985a. Synthesizing distributed and parallel programs through optimistic transformations. In *Proceedings of the 1985 International Conference on Parallel Processing* (St. Charles, Ill. Aug. 20–23). Penn State University, pp. 632–641.

STROM, R. E., AND YEMINI, S. 1985b. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst. 3*, 3 (Aug.), 204–226.

STROM, R. E., AND YEMINI, S. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng. SE-12*, 1 (Jan.), 157–171.

STROM, R. E., YEMINI, S., AND WEGNER, P. 1985. Viewing Ada from a process model perspective. In *Proceedings of the Conference on Ada in Use* (Paris, France, May 14–18). ACM, New York.

STROUSTRUP, B. 1986. *The C++ Programming Language.* Addison-Wesley, Reading, Mass.

SUGIMOTO, S., AGUSA, K., TABATA, K., AND OHNO, Y. 1983. A multi-processor system for Concurrent Lisp. In *Proceedings of the 1983 International Conference On Parallel Processing* (Bellaire, Mich.). pp. 135–143.

SWINEHART, D. C., ZELLWEGER, P. T., AND HAGMANN, R. B. 1985. The structure of Cedar. In Proceedings of ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments. *SIGPLAN Not.* (ACM) *20*, 7 (July), 230–244.

TAKEUCHI, A., AND FURUKAWA, K. 1985. Bounded buffer communication in Concurrent Prolog. *J. New Generation Comput. 3*, 2, 145–155.

TAKEUCHI, A., AND FURUKAWA, K. 1986. Parallel logic programming languages. In *Proceedings of the 3rd International Conference on Logic Programming* (London, July 14–18). Springer-Verlag, Berlin, pp. 242–254.

TANENBAUM, A. S. 1987. *Operating Systems: Design and Implementation.* Prentice-Hall, Englewood Cliffs, N.J.

TANENBAUM, A. S., AND VAN RENESSE, R. 1985. Distributed operating systems. *ACM Comput. Surv. 17*, 4 (Dec.), 419–470.

TANENBAUM, A. S., AND VAN RENESSE, R. 1988. A critique of the remote procedure call paradigm. In *Proceedings of the EUTECO 88 Conference* (Vienna, Austria, Apr. 20–22). North-Holland, Amsterdam, pp. 775–783.

TANENBAUM, A. S., VAN STAVEREN, H., KEIZER, E. G., AND STEVENSON, J. W. 1983. A practical toolkit for making portable compilers. *Commun. ACM 26*, 9 (Sept.), 654–660.

TAYLOR, S., AV-RON, E., AND SHAPIRO, E. 1987a. A layered method for process and code mapping. *J. New Generation Comput. 5*, 2, 185–205.

TAYLOR, S., SAFRA, S., AND SHAPIRO, E. 1987b. A parallel implementation of Flat Concurrent Prolog. *Int. J. Parallel Program. 15*, 3, 245–275.

TRELEAVEN, P. C., BROWNBRIDGE, D. R., AND HOPKINS, R. P. 1982. Data-driven and demand-driven computer architectures. *ACM Comput. Surv. 14*, 1 (Mar.), 93-143.

TSUJINO, Y., ANDO, M., ARAKI, T., AND TOKURA, N. 1984. Concurrent C: A programming language for distributed systems. *Softw. Pract. Exper. 14*, 11 (Nov.), 1061-1078.

UEDA, K. 1985. Guarded Horn clauses. ICOT TR-103, Institute for New Generation Cómputer Technology, Tokyo, Japan, June.

VISHNUBHOTIA, P. 1988. Synchronization and scheduling in ALPS objects. In *Proceedings of the 8th International Conference on Distributed Computing Systems* (San Jose, Calif., June 13-19). IEEE, New York, pp. 256-264.

WEIHL, W., AND LISKOV, B. 1985. Implementation of resilient, atomic data types. *ACM Trans. Program. Lang. Syst. 7*, 2 (Apr.), 244-269.

VAN WIJNGAARDEN, A., MAILLOUX, B. J., PECK, J. E. L., KOSTER, C. H. A., SINTZOFF, M., LINDSEY, C. H., MEERTENS, L. G. L. T., AND FISKER, R. G. 1975. Revised report on the algorithmic language Algol 68. *Acta Inf. 5*, 1-236.

WILKES, C. T., AND LEBLANC, R. J. 1986. Rationale for the design of Aeolus: A systems programming language for an action/object system. In *Proceedings of the IEEE CS 1986 International Conference on Computer Languages* (Miami, Fla., Oct.). IEEE, New York, pp. 107-122.

WILKES, C. T., AND LEBLANC, R. J. 1988. Distributed locking: A mechanism for constructing highly available objects. In *Proceedings of the 7th Symposium on Reliable Distributed Systems* (Columbus, Ohio, Oct. 10-12). IEEE, New York.

WIRTH, N. 1971. The programming language Pascal. *Acta Inf. 1*, 35-63.

XU, A. S. 1988. A fault-tolerant network kernel for Linda. Tech. Rep. 424, MIT Laboratory for Computer Science, Cambridge, Mass., Aug.

YANG, R. 1988. *P-Prolog: A Parallel Logic Programming Language.* World Scientific Publishing Co., Singapore.

YANG, R., AND AISO, H. 1986. P-Prolog: A parallel logic language based on exclusive relation. In *Proceedings of the 3rd International Conference on Logic Programming* (London, July 14-18). Springer-Verlag, Berlin, pp. 255-269.

YEMINI, S. 1982. On the suitability of Ada multitasking for expressing parallel algorithms. In *Proceedings of the AdaTec Conference on Ada* (Arlington, Va., Oct. 6-8). ACM, New York, pp. 91-97.

YOKOTE, Y., AND TOKORO, M. 1986. The design and implementation of ConcurrentSmalltalk. In Proceedings of Object-Oriented Programming Systems, Languages and Applications 1986. *SIGPLAN Not.* (ACM) *21*, 11 (Nov.), 331-340.

YOKOTE, Y., AND TOKORO, M. 1987a. Concurrent programming in ConcurrentSmalltalk. In *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, Eds. MIT Press, Cambridge, Mass., pp. 129-158.

YOKOTE, Y., AND TOKORO, M. 1987b. Experience and evolution of ConcurrentSmalltalk. In Proceedings of Object-Oriented Programming Systems, Languages and Applications 1987. *SIGPLAN Not.* (ACM) *22*, 12 (Dec.), 406-415.

YONEZAWA, A., BRIOT, J.-P., AND SHIBAYAMA, E. 1986. Object-oriented concurrent programming in ABCL/1. In Proceedings of Object-Oriented Programming Systems, Languages and Applications 1986. *SIGPLAN Not.* (ACM) *21*, 11 (Nov.), 258-268.

ZHONGXIU, S., AND XINING, L. 1987. CSM: A distributed programming language. *IEEE Trans. Softw. Eng. SE-13*, 4 (Apr.), 497-500.

Received June 1988; final revision accepted April 1989.