1 INTRODUCTION

Simulation is a problem solving technique that involves modeling a dynamic system and observing its behavior over time. A system may be defined as a collection of inputs which pass through certain processing phases to produce outputs. A manufacturing system may use crude oil as an input and a cracking plant for processing that produces various types of oil and gasoline as outputs. The first step in studying a system is to build a model. A model can be a formal representation of theory or empirical observation. Usually it is a combination of both. Simulation enables experimentation with systems which do not yet exist, or for which it is difficult to get first hand experience. It also enables repeated experimentation with a system, under controlled conditions, to optimize performance. From the simulationists point of view a model can be constructed to focus on important aspects of the problem and other, less important, details can be ignored. A fundamental problem in simulation is validation, that is ensuring that the model behavior mimics the behavior of the system being investigated with acceptable accuracy.

Three classes of simulation can be defined: discrete, continuous and combined. Discrete event simulation refers to a modeling technique that enables instantaneous changes in the state of a model to be made at discrete points in time. Continuous simulation implies that changes in state occur smoothly and continuously in time, i.e., models usually involve differential equations. Combined simulation is a technique which simulates systems owning both discrete and continuous characteristics. Generally practical simulation work involves the following steps:

(1) ***Definition of the problem and simulation objectives*** such as what questions need to be answered, what aspects of system behavior need to be modelled, what results or performance measures need to observed as output, etc.

(2) ***Model specification and collection of data.*** A system can be modeled only after its components and interactions have been isolated and defined. Model specification can be defined as the correct description of an internal representation and a set of transformation rules which can be used to predict the behavior and relationships among the set of entities composing the system. Data should be collected on the system of interest and used to estimate input parameters and to obtain probability distributions for the random variables used in the model.

(3) ***Construction of a computer program.*** Operation of the system model is represented by the execution of a program written in some programming language. A good simulation language may reduce the required programming effort significantly and also lead to a shorter simulation execution time.

(4) ***Verification and validation of the simulation program and model.*** Verification involves determining whether a simulation model performs as intended, i.e., the computer program must be debugged. Traces are a useful tool for relating the behavior of the simulation program to the specification of the model. Validation involves determining whether a simulation model (as opposed to the computer program) is an accurate representation of the system under study.

(5) ***Simulation experiments and analysis.*** Production runs are made to provide performance data on the system designs of interest. Statistical techniques are used to analyze the output data from the production runs. Typical goals are to construct a confidence interval for a measure of performance for one particular system design or to decide which simulated system is best relative to some specified measure of performance.

(6) ***Documentation and implementation of the results.*** Because simulation models are often used for more than one application, it is important to document the assumptions, inputs, outputs, results and conclusion drawn from a set experiments. Since simulation is a decision making tool, the analysis of simulation results usually leads to specific recommendations.

Simulation currently involves sequential program execution primarily because of the wide and intensive sharing of variables (e.g., time) by all model components. However, interesting classes of systems exhibit a high degree of natural parallelism, i.e., they can be decomposed into a set of concurrently operating objects. For example, distributed computer systems, traffic control systems, banking systems

and most daily human activities. This fact and the emergence of highly parallel, distributed computer systems, has led many scientists to attempt distributed simulations that involve the concurrent execution of model components. A commonly used technique for distributed simulation is the decomposition of a model into communicating, sequential processes, where processes communicate only through message passing. There are no shared variables and there is no central process that provides message routing or process scheduling. The goals of distributed solutions are to speed up simulation by exploiting the parallelism inherent in many real systems and their models, and to get more accurate measurement of parallel behaviour.

In this paper, we are interested in process communication mechanisms and programming languages which support distributed, discrete event simulation. These topics are addressed in the following two sections, respectively. Finally, we conclude that a Prolog-like language in conjunction with the Time Warp mechanism offers potential advantages over other approaches.

2. COMMUNICATION MECHANISMS

Traditionally, discrete-event simulation is performed by maintaining an **event list** which is used to synchronize and schedule different cooperating activities. In a distributed simulation there is no central controller. Once a simulation model has been decomposed into processes, one has to solve the problems of synchronizing these processes so that events occur in a correct order. The first key notion is simulation time, which plays a special logical role. The concept of time is fundamental to determine the order in which events occur. Methods of distributed simulation using a central clock have been proposed[1][2]. However more attractive suggestions use distributed clocks[3][4][5][6]. The other key notion is inter-process communication, which can be broadly divided into two classes, synchronous and asynchronous methods. With synchronous methods all processes in the simulation progress forward in simulation time together, in synchrony, with no process ahead in time of any other. In contrast, an asynchronous methods permit some processes to go ahead in time while others lag behind. In any case, there must be some mechanism for ensuring that all the events in a simulation are processed in correct order. Asynchronous methods try to exploit the maximum parallel execution of processes within a simulation model. An interesting relationship (not necessarily for all the cases) has been seen between central clock schemes and synchronous methods and between distributed clocks schemes and asynchronous methods. Chandy[7] uses two terms to distinguish these systems. The former is called "Time driven simulation system" and the later "Time exchange simulation system". In the rest of this section, we restrict discussion to the time exchange simulation systems.

2.1. LOGICAL CLOCKS

The proverb: "A person with one watch knows what time it is; a person with two or more watches is never sure." is commonly accepted, e.g., Lamport[3]. He notices that it is sometimes impossible to say that one of two events occurred first in a distributed system: "...Most people would probably say that an event **a** happened before an event **b** if **a** happened at an earlier time than **b**. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time."

Lamport carefully reexamined the events in a distributed system and found that real-time temporal order, simultaneity, and causality between events bear a strong resemblance to the same concepts in special relativity. From the "space -time diagram" in his paper, he showed that the real-time temporal relationships "happens before" and "happens after" only form a partial order. So he introduced logical clocks to extend this partial order to a total order. Being able to totally order the events can be very useful in implementing a distributed system.

Each process has an associated logical clock which is used to assign a number to an event, where the number is thought of as the timestamp at which the event occurred. A process can execute a command timestamped T when it has learned of all commands issued by all other processes with timestamps less than or equal to T. This method allows one to implement any desired form of multiprocess synchronization in a distributed system. However, there are two shortcomings. The first problem is that there is a large volume of communication traffic among processes because each process has to broadcast its request to all other processes, and receive the same amount of acknowledgements when it executes a command related to synchronization. Second, a process must know all the commands issued by other processes, so that the failure of a single process will be fatal.

Although Lamport's work did not concentrate on distributed discrete-event simulation, it does help in understanding the basic problems of multiprocess systems and the importance of logical clocks.

2.2. THE NETWORK PARADIGM

Most past work on distributed simulation has been based on the network paradigm[4][5][7][8]. In this paradigm, objects are represented as a collection of parallel processes acting as nodes in a network. Each directed line in the network represents an one way communication channel between two processes (nodes). The crucial issue in distributed simulation is that of simulating time on multiple processes. Chandy[7] suggests using "time exchange" method instead of using a central clock which drives all the processes. Each process has to associate with it a local clock which moves forward in time in an asynchronous manner, and tells how far the associated process has progressed in the simulation. The key points of their method are the following:

(1). There is a separate clock (time) associated with each line connecting processes (nodes).

(2). Processes communicate only by passing messages that include time stamps.

(3). Each process attempts to move the output clocks as far ahead in time as possible, based upon currently available information.

(4). The output message on a line may state that no event will arrive on that line between the current clock-time and some future time. The use of a no-event message is crucial to the correct operation of distributed simulators.

(5). Since the sequence of clock times on a line are monotone increasing, merging of two lines at a process can be achieved using the well known merging algorithm.

Idealy, if each process (node) in the simulation were assigned to a different processor and all could execute concurrently, then we would achieve an optimal n-fold speedup over the single processor case. Unfortunately, it is not always true in real simulations. For example, frequently a process will have one or more of its input lines with no currently available information (empty queue) when it tries to receive the next message. When this happens it must wait until all of its input lines are not empty, and then select the next message with correct time order. In this case, two severe problems might arise, that is, deadlock and memory over- flow problems. Chandy proposed a mechanism that directly deals with both problems[5]. First, he requires that each message queue have a bounded length. In addition to blocking whenever one of its input line is empty, a process must also block whenever it sends a message along an output line where the message queue at the other end is full. Second, the distributed simulations always run with a deadlock detection algorithm, as soon as a deadlock situation has been detected, a deadlock

breaking algorithm is activated.

The network paradigm proposed a distributed solution for simulation problems which are typically solved in a sequential manner. It showed that the time required to run a distributed program for the specific problem of queuing network simulation is generally less than the time required to run corresponding sequential programs. This efficiency is achieved since there is no global process which could be a bottleneck. However, the concurrency is limited by blocking when the input queue is empty or the output queue is full. The technique for solving the memory overflow problem exacerbates the deadlock problem. Since a process can block while either sending or receiving, a deadlock situation can occur around any undirected cycle in the simulation network, rather than just in the directed cycles.

2.3. THE TIME WARP MECHANISM

It sometimes makes sense to have processors performing computations that may never be needed if indeed the possibility that they are used will speed the execution of a given computation. This idea is called the "Never Wait Rule" which states that it is better to give a processor a task that may or may not be used than it is to let the processor sit idle. Before Jefferson's innovative work[6][9], nearly all the proposals for distributed simulation involved conservative synchronization mechanisms. In a word, conservative mechanisms involve "waiting" - waiting for synchronization of sender and receiver, or waiting to make decisions that correctness can be proved at some check points. In other words, these proposals do not obey the "Never Wait Rule". The Time Warp mechanism never involves waiting. Jefferson uses the term Virtual Time in connection with the Time Warp mechanism to be synonymous with simulation time and describes the main idea as follows: "...A Virtual Time system is a distributed system that executes in coordination with an imaginary global virtual clock that ticks virtual time. Virtual time is a temporal coordinate system used to measure computational progress and define synchronization. increasing virtual receive time order as long as it has any messages left. All of its execution is provisional, however, because it is constantly gambling that no message will ever arrive with a virtual receive time less than the one stamped on the message it is now processing. As long as it wins this bet execution proceeds smoothly. The novelty is that whenever the bet is lost the process pays by rolling back to the virtual time when it should have received late message. The situation is quite similar to the gamble that paging mechanisms take in the implementation of virtual memory: They are constantly betting with every memory reference that no page fault will occur. Execution is smooth as long as the bet is won, but a comparatively expensive drum read is necessary when it is lost."

In the Time Warp paradigm, each process always charges ahead, blocking only when its input queue is exhausted, and then only until another messages arrives. Whenever a message with a timestamp "in the past" arrives at a process's input queue, the Time Warp mechanism automatically restores that process to a state from a virtual time earlier than the timestamp of the late message, cancels any side effects that it may have caused in other processes, and then starts the process forward again. Although some computational effort is "wasted" when a projected future is thrown away, a conservative mechanism would keep the process blocked for the same amount of time, so the time would be "wasted" anyway. According to the Never Wait Rule, the Time Warp mechanism can speed up almost any large simulation by exploiting the concurrency within it.

A question here is: to what extent we can win via this type of gambling? If we always lose, even if we do not waste time on synchronizations, we have to pay a lot of costs for rollback actions. Like the paging systems, the Time Warp mechanism is also based on the Locality Assumption. Locality manifests itself in both time and space. Temporal Locality is locality over time. Spatial locality means that nearby items tend to be simular. Locality is observed in operating system environments, particularly in the area of storage management. It is an empirical property rather than a theoretical one. It is never guaranteed but is often highly likely. Actually, locality is quite reasonable in distributed simulation systems, when

one considers the way programs are written and communication is organized. In particular, temporal locality means that if process A sends a message to process B, it is most likely that process A sends another message to, or receives a reply from process B in the near future, and that such subsequent messages will have increasing timestamps. For example, the communication between an event generating process and an event consuming process. Spatial locality means that the communication connections between processes are most likely stable. The success of the Time Warp mechanism is based on this locality assumption, even though it has not been tested experimentally.

In this section, we outlined the fundamentals of the Time Warp mechanism. Further details can be found in [6][9]. It appears that the Time Warp mechanism is an attractive paradigm for distributed simulation. It is deadlock free, completely transparent to the programmer, and seems to have significant advantages over conservative mechanisms. However, there are problems which need further exploration. One problem is the state-saving and rollback mechanisms. In the case of rollback, the rollback mechanism must hold a state queue which saves copies of the process's past states, ordered by the local virtual time. State-saving is a low level system activity. In general, what should be saved and what should not, is not known, as Jefferson discussed in [6]. The entire data space of a process is saved each time, which is not only space-consuming, but also time-consuming.

3. DISTRIBUTED PROGRAMMING LANGUAGES

Traditionally, programming languages for discrete-event simulation are sequential or pseudo-concurrent languages. Discrete systems generally involve contention for scarce resources, with queues developing where system components must wait for resources to become available. Further, delays between state changes are usually determined statistically, with the exact interval selected according to some random number distribution. Some of the more popular discrete simulation languages are GPSS[10], SIMULA[11], and SIMSCRIPT[12]. These kinds of languages are procedural, i.e., a program explicitly specifies the steps which must be performed to reach a solution. Another kind of language has recently been used in simulation, i.e., T-Prolog[13]. It is a declarative or descriptive language, i.e., it is only necessary to describe the problem in terms of statements and rules that define relationships among the objects in question.

Recently, many proposals have been put forward for distributed programming, including CSP[14], DP[15], PLITS[16], E-CLU[17], *MOD[18], Cell[19], Soma[20], NIL[21], ADA[22], PARLOG[23] and Concurrent Prolog[24]. These languages all support concurrent computation. One issue for distributed systems is designating which actions can be executed concurrently. Almost all these languages use explicit processes to show concurrency, although they may use different names for the same concept. In this section, we shall briefly survey these languages by dividing them into two groups, i.e., the procedural and declarative programming languages.

3.1. THE PROCEDURAL PROGRAMMING LANGUAGES

The distributed procedural languages inherit most features from conventional programming languages. These include support for abstraction, particularly abstract objects, support for modularization, including separate compilation of modules, support for sequential execution flow control, support for strong data types and data encapsulation, and support for error and exception handling. However, as a distributed program resides and executes at communicating, but geographically distinct, nodes of a network, a distributed programming language must provide the functions of distribution and communication. The latter constitute the major difference from conventional programming languages.

Distribution means process dynamics which describes the change in number and variety of processes through the execution of a distributed program. Two methods are commonly used to create new processes. Some languages allow programs to create new processes during execution (dynamic processes). The syntactic mechanisms supporting dynamic process creation are explicit allocation and lexical program elaboration. Languages with explicit allocation have a statement to create a new process, such as Ada, NIL and PLITS. Lexical elaboration creates processes by combining declarations with recursive program structures. That is, if procedure P declares process A and then calls itself recursively, the recursive invocation of P creates another copy of A. Cell, Ada, and PLITS create new processes by lexical elaboration. Another method is that languages require that all processes be defined at system creation (static processes), such as CSP, DP, *MOD, and Soma. The dynamic process creation is more flexible than the static one, but the later is easier to understand.

The differences that distinguish distributed programming languages from sequential ones center on communication and synchronization among processes. As we discussed in Section 2, the key notion among languages is the issue of synchronous versus asynchronous communication. In a synchronous scheme, every communication request is matched by a reception; a process cannot send the second message until the first one has been handled. In an asynchronous scheme, processes send messages without regard to their reception; a process is free to send a message and continue computing. PLITS and Soma use an asynchronous communication scheme. NIL provides both synchronous and asynchronous communication. The rest adopt synchronous communication. Other issues in distributed programming languages are communication connection and message control. These two issues have a close relationship for establishing communication among processes. Communication connection is a naming problem. Three different syntactic forms - ports, names and entries, are used to channel communication. Communication through a special typed symbol is communication through a port. A port can be referenced by communicating processes through global declaration or ownership transfer. *MOD, PLITS, NIL, Soma and E-CLU use ports (possibly using other names such as "mailbox"). Several languages - Ada, Cell, and DP, focus communication on an entry in the called process. A called process can have several entries and accept requests from them in an order determined by program control. CSP uses process names to communicate directly. In order to exchange messages, two processes must identify each other by their names in input and output statements. In this case, even though the communication connection looks explicit, the lack of anonymous communication makes it difficult to build program libraries. Message control concerns the actions that processes take to communicate, including the facilities they have for choosing a communication partner and segregating incoming messages. For example, CSP treats processes as equals. It introduces asymmetric unidirectional message flow. Input guards provides concurrency control. Alternative commands combined with input guards can automatically segregate incoming messages. Other languages specify roles for the "calling" and "called" processes. Ada, Cell and PLITS allow the called process some freedom in choosing which request to serve. All incoming messages are segregated into groups by entry queues.

Some discrete event simulation systems have already been built based on some of these languages[1] [25]. Since they adopt synchronous communication and a central clock for simulation time, additional work is required to exploit the concurrency of the parallel simulation model.

3.2. THE DECLARATIVE PROGRAMMING LANGUAGES

A declarative programming language is a logic programming language in the sense that its statements are interpreted as sentences of a logic. The ancestor of these languages is Prolog which was developed about 10 years ago. Prolog has been very popular in Europe and is now targeted as the core language of the Japanese Fifth Generation Computer Project. Several advantages to the use of Prolog-like languages for simulation are summarized as follows[26]:

(1) They provide declarative semantics based on logic in addition to the usual procedural semantics.

(2) Program and data are identical in form and can therefore be easily manipulated.
(3) Arguments of procedures are not fixed as input or output parameters as in other programming languages and procedures may have multiple inputs and outputs.
(4) Backtracking is used to find a complete set of solutions for a given problem. This also results in a high-level form of iteration.
(5) The basic elements (atoms, variables and compound terms) provide a general and flexible data structure superior to the arrays and records used in conventional programming languages.
(6) The language designs are well suited to parallel search and are, therefore, excellent candidates for future powerful computers incorporating parallel processing.
(7) Programs are usually significantly shorter than programs written in imperative programming languages (typically 5-10 times shorter).

T-Prolog is an extension of Prolog to define a goal oriented discrete simulation. It provides facilities similar to those found in conventional simulation languages. The basic features in T-Prolog are the notions of process and the system internal clock. T-Prolog adopts dynamic process creation and allows processes communicating with each other during their execution. The attraction of T-Prolog is the use of backtracking to automatically modify the model until the simulation exhibits some desired behaviour. However, a problem with T-Prolog is that all processes are driven by a central clock which acts as a bottleneck in a distributed system.

Two other varieties of Prolog are PARLOG[23] and Concurrent Prolog[24]. They are parallel logic programming languages featuring both AND-parallelism and OR-parallelism. PARLOG relations are divided into two types: single-solution relations and all-solutions relations. A conjunction of single-solution relation calls can be evaluated in parallel with shared variables acting as communication channels for the passing of partial bindings. Only one solution to each call is computed, using committed choice nondeterminism. A conjunction of all-solutions relation calls can be evaluated without communication of partial bindings, but all the solutions may be found by an or-parallel exploration of the different evaluation paths. PARLOG uses mode declarations to determine the communication constraints on processes. Concurrent Prolog is very similar with the single-solution component of PARLOG. They both use guarded clauses, committed choice nondeterminism, and the ability to have variables in messages. The major difference is in the way the interprocess communication constraints are expressed. In Concurrent Prolog, programs do not have fixed modes of use. Instead, variables shared between goals serve as the process communication mechanism.

The first implementations of PARLOG and Concurrent Prolog have been built on Prolog. Current implementation efforts are directed toward both parallel and conventional sequential machines. It seems that PARLOG and Concurrent Prolog have a very definite advantage in execution speed, but this depends on the emergence of suitable parallel processing machines.

4. DISCUSSION

Practical simulation work involves defining a project and its goals, specifying the model, implementing it as a working computer program, verifying and validating, experimenting with the model, and producing documentation. The development of a large simulation model is a complex and difficult task. In many research areas, expensive computers are devoted almost exclusively to simulation. It is therefore becoming an increasing important research goal to speed up simulations by exploiting the concurrency inherent in them.

The interprocess communication mechanisms and programming languages discussed in this paper span the important ideas for distributed simulation. When we seek an ideal distributed discrete simulation language system, we are drawn to the following questions: What set of communication primitives can best describe the run-time behavior of our system? What interprocess communication mechanisms built from these primitives best support distributed simulation? What language constructs are suitable to describe simulation models which can be efficiently implemented with these primitives while leaving the programmer unaware of the lower level run-time environment? Of course, we hope that this system has: the flexibility that comes from dynamic process creation, the concurrency that comes from asynchronous communication, the explicit naming facility that not only makes a program easy to understand but also makes it easier to develop large programs and libraries, and powerful expressive facilities that can decrease development time.

Comparing different interprocess communication mechanisms, we focus our attention on the Time Warp mechanism. Based on the Never Wait Rule and the Locality Assumption, the Time Warp mechanism can speed up almost any large simulation by exploiting the concurrency within it. Clearly, discrete simulation is one of the most appropriate applications of the Time Warp paradigm, because the virtual times (simulation times) of events are completely under the control of the user. However, the simulation programmer does not need to know that his program is running under the Time Warp mechanism. He uses the same conceptual methodology for building a concurrent, object-based simulation as he uses for sequential, object-based simulation. These advantages are not without cost. As Jefferson [6] expected, the Time Warp mechanism uses several times as much memory as other methods in order to achieve maximum speed up. The major memory cost is due to state saving. The Time Warp mechanism has no knowledge about what should be saved and what should not, and thus the entire data space of a process is saved. If a process manipulates a large amount of data, say, a matrix with 100*100 integers, the memory will be exhausted very soon by saving several successive states. One may argue that the cost of state saving can be reduced if only a few own variables that represent the whole state are saved instead of saving the entire data space. In this case, the responsibility is placed on the user. The programmer has to isolate a set of representative variables from his own data domain, then present them through some linguistic declarations to the Time Warp system. The defect of this scheme is the destruction of transparency.

Re-examining the programming languages discussed in Section 3, we found that the backtracking facility of a Prolog-like language offers a possible way to overcome the shortcoming of non-knowledgeable state saving. When Prolog rolls back and re-satisfies a goal, it returns to the most recently instantiated variables and attempts to instantiate them with alternative values. If this is not possible it backs-up further to the next most recently instantiated variables, etc. Thus, the set of variables changed on a given computation path is completely known. This knowledge can be used to determine the exact amount of information that must be saved for any given execution path. In other words, state saving in Prolog is based on knowledge. Therefore, a Prolog-like language combined with the Time Warp mechanism not only reduces the cost of state saving, but also keeps the system completely transparent.

In addition, having its foundation in logic, a Prolog-like language encourages the programmer to describe problems in a manner that facilitates checking for correctness and consequently reduces the verification, i.e., the debugging, effort. A Prolog-like language can be used as a tool both for model specification and program implementation. Since Prolog provides general, flexible data structures and procedures which may have multiple inputs and outputs of dynamically defined types, we do not need any extra facilities such as entries, ports, or alternative commands for sorting and segregating incoming messages. For example, we can write a universal "Queue" process in a Prolog-like language, which can direct messages with arbitrary data structures between senders and receivers. We can write a message processing procedure with different arguments (set of clauses) and Prolog will automatically search for the matching clause. Thus, we believe that a Prolog-like programming language in conjunction with a run-time kernel based on the Time Warp mechanism offers potential advantages over other approaches.

A programming environment that supports the development of distributed software, i.e. Jade, was developed during 1982-85 that is now in use at a number of universities and research institutes. Jade provides an integrated set of tools for monitoring, debugging and graphically animating the execution of distributed programs. Early in 1985 we began work on a version of Time Warp that is based on the Jade environment which has now been completed. A major goal of this Jade Time Warp system is to enable studying different mechanisms for state saving and rollback within the context of different programming languages. Our next step is to develop a Time Warp Prolog system that uses Prolog backtracking to accomplish rollback.

REFERENCE

[1]. W. H. Kaubisch and C. A. R. Hoare,
"Discrete event simulation based on communicating sequential processes"
[2]. K. Broda and S. Gregory,
"PARLOG for discrete event simulation" Mar, 1984, Research report DOC 84/5, University of London
[3]. L. Lamport
"Time, clocks, and the ordering of events in a distributed system." CACM 21,7, July, 1978
[4]. K. M. Chandy and J. Misra,
"Asynchronous distributed simulation via a sequence of parallel computations" CACM 24,4, Apr. 1981
[5]. K. M. Chandy and J. Misra,
"Distributed simulation: A case study in design and verification of distributed programs" IEEE Software Eng., SE-5, 5, Sept. 1979
[6]. D. R. Jefferson
"Virtual Time" CACM 7,3, July, 1985
[7]. K. M. Chandy, Victor Holmes, and J. Misra,
"Distributed simulation of networks" Computer Networks, Vol. 3, No. 2, 1979
[8]. J. K. Peacock, J. W. Wong, and E. G. Manning,
"Distributed simulation using a network of processors" Computer Networks, Vol. 3, No. 2, 1978
[9]. D. R. Jefferson and H. A. Sowizral,
"Fast concurrent simulation using the Time Warp mechanism" Proc. of the SCS distributed simulation conference, Jan., 1985
[10]."General Purpose Simulation System V User's Manual" IBM Corporation, White, Plains, N. Y. ,1977
[11].G. M. Birtwistle,
"Discrete event modeling on Simula" Macmillan Computer Science Series, 1979
[12].G. S. Fishman,
"Principles of Discrete event simulation" Wiley Series on Systems Engineering and Analysis, 1978
[13].I. Futo and J. Szeredi,
"T-Prolog: A very high level simulation system" 1982
[14].C. A. R. Hoare,
"The communicating sequential processes" CACM, Vol.21, Aug., 1978,
[15].P. B. Hansen,
"Distributed processes: a concurrent programming concept" CACM, Vol.21, Nov., 1978,
[16].J. A. Feldman,
"High level programming for distributed computing" CACM, Vol.22, June, 1979,
[17].B. Liskov,
"Primitives for distributed computing", Proc. of the 7th symposium on operating system principle, 1979,
[18].R. P. Cook,
"*MOD - A language for distributed programming", IEEE SE-6, No.6, Nov., 1980,
[19].A. Sillberschatz,
"A note on the distributed program component cell", ACM SIGPLAN NOTICES, Vol.16, No.7, July,

1981,
[20]. J. L. W. Kessels,
"The Soma: a programming construct for distributed processing", IEEE SE-7, No.5, Sep., 1981,
[21].R. Strom and S. Yemini,
"The NIL distributed systems programming language: a status report", ACM SIGPLAN Notices, V20, #5, May, 1985,
[22]."Reference Manual for the Ada programming language", United States DOD, July, 1980,
[23].K. Clark and S. Gregory,
"PARLOG: parallel programming in logic", ACM Tras. on programming languages and systems, Vol.8, No.1, Jan., 1986,
[24].E. Y. Shapiro,
"A subset of Concurrent Problem and its interpreter", Tech. rep. TR-003, ICOT, Tokyo, Feb., 1983,
[25].B. W. Unger, G. A. Lomow, and G. M. Birtwistle,
"Simulation software and Ada", SCS, A publication of The Society for Computer Simulation, 1984
[26].R. E. Shannon, R. Mayer, and H. H. Adelsberger,
"Expert systems and simulation", SIMULATION, June, 1985