Object-Oriented Distributed Simulation

Joel West

CACI

Revised Edition
August 1985

This work was performed for the Jet Propulsion Laboratory, California Institute of Technology, contract number 957101, and was sponsored by the Army Model Improvement Program Management Office.

Object-Oriented Distributed Simulation

Joel West

CACI

Revised Edition
August 1985

This work was performed for the Jet Propulsion Laboratory, California Institute of Technology, contract number 957101, and was sponsored by the Army Model Improvement Program Management Office.

ABSTRACT

This report examines the requirements for an object-oriented, discrete event simulation language for use in a parallel-processing environment. Such a language could be used to distribute computation and data on systems such as the Caltech Hypercube and JPL's Time Warp operating system.

The report first summarizes a number of existing and planned multiple instruction set, multiple data stream (MIMD) computers. The report also outlines the general principles for object-oriented programming and distributed simulation.

A proposed language that incorporates these requirements is then described. The proposed language Language for Concurrent Simulation has some of the key characteristics of Smalltalk-80 and the Lisp Flavor System. The language is unique, to the author's knowledge, in providing both class- and instance-oriented inheritance of object behaviors in a compiled language. It also includes the SIMSCRIPT process model of object behaviors as part of its conceptual framework.

PREFACE

This report is the result of a study conducted to determine the characteristics of a discrete simulation language for use in parallel processing.

The study is an outgrowth of parallel-processing research being done at the Jet Propulsion Laboratory for the Army Model Improvement Program Management Office. That research -- into the Hypercube computer and Time Warp operating system -- is intended to allow the Army's operations research groups to obtain high simulation performance at a low hardware cost.

This report also attempts to meet a second goal of the AMMO-sponsored research, which is to identify and describe a new generation of simulation tools for the development of large production combat models. The author is convinced that the language described herein is both feasible and highly desireable for simulation modelling.

CACI would like to thank JPL for its decision to sponsor this study. We also thank Jack Tupman and Dr. Garrett Paine of JPL for their valuable analysis of our preliminary results. Fred Wieland of JPL patiently explained the details of the current implementation Time Warp operating system.

Special thanks are due to Professor David Jefferson of UCLA for providing information on Time Warp concepts and discussing his ideas on distributed simulation.

CACI also wishes to thank AMMO for its interest in the study. Without the encouragement of Harry Jones and Col. Kenneth Wiersema, this study would never had taken place.

Thanks go to Dr. Wilbur Payne and Jesus Carillo, of the TRADOC Operations Research Activity, for guidance on the Army's future simulation plans. Also, Jim Peters and members of the Simulation and Computer Support Division of the TRADOC Systems Analysis Activity provided valuable information about existing Army simulations.

Dr. Ed Russell and Glen Johnson of CACI provided assistance from their wealth of experience in simulation and simulation languages. Dr. Alasdar Mullarney of CACI N.V. slipped away from pressing deadlines to sharpen some of the details contained herein.

Finally, I would like to offer my sincerest thanks to Professor Antonio Elias and Dr. John Pararas of the Massachusetts Institute of Technology. Besides sharing their valuable expertise in Smalltalk and the Lisp Flavor System, Chapter 3 draws heavily from their work **[Elias 1985]**. The ongoing dialogue in the language design proved invaluable, and the concept of multiple inheritance in the proposed language owes its existence to their persistent efforts.

Joel West
La Jolla, California
August 5, 1985

TABLE OF CONTENTS

Table of Contents

Chapter 4: A Language for Concurrent Simulation

Chapter 5: Distributed Simulation using LCS

Chapter 6: Conclusions and Recommendations

Table of Contents

CHAPTER 1: BACKGROUND FOR THE REPORT

1.1 The Army Model Improvement Program

Reflecting its significant investment in discrete-event simulation, in 1983 the Department of the Army issued a revised Army Regulation 5-11, detailing the Army Model Improvement Program. As specified in **[Army 1983]**, the regulation provides for the Army Model Improvement Program Management Office (AMMO) to coordinate efforts to improve the effectiveness of the Army's simulation groups.

Through research in both performance and productivity, AMMO hopes to meet these objectives:

* Increase productivity in model development
* Easier maintenance of major models
* Improved capability for modeling complex systems
* Develop capability to run large models faster

One of the current problems is that, at the current level of complexity, simulation scenarios are currently far too slow for the available computational resources. For example, the CASTFOREM model currently requires 20 hours of dedicated VAX-11/780 computer time ("cpu time") per simulated engagement. Twenty such engagements are required per scenario for statistical validity.

Current research in computer hardware suggests that the performance of existing processing units can be significantly increased only at exceptional cost. Many researchers feel that cost-effective increases in raw computing power should instead be sought by developing hardware and software to support parallel-processing techniques.

This approach would appear to offer great promise for large military simulations. In general, the systems being modeled have a large number of parallel actions -- by tanks, planes, trucks, men, etc. -- that will also be found in the execution of the simulation.

Object-Oriented Distributed Simulation

Table 1-1 lists pertinent information on a number of significant Army-sponsored models, including three models in the current AMIP plan: FORCEM, CORDIVEM, and CASTFOREM. The table shows the number of objects in each simulation as an indication of potential concurrency. It also lists each model's size, to indicate the minimum hardware resources necessary to support the model.

Agency	Model	Size of model Lines	Size of model Memory	Number of active objects
TRASANA	CASTFOREM	200,000	20 mb	700
	VIC	40,000	n.a.	400
CAORA	CORDIVEM	200,000	15 mb	2,500
CAA	FORCEM	110,000	6 mb	3,000
CAA/ARMTE	DEWCOM	23,000	5 mb	800
CAA/USAWC	JTLS	120,000	25 mb	300-2,500

Table 1-1: Major Army-sponsored models

In Fiscal Year 82-83, the Army Model Improvement Program Management Office (AMMO) began funding research to explore promising techniques for exploiting this concurrency through the use of parallel-processing computers. This research has largely taken place at the Jet Propulsion Laboratory in Pasadena, Calif.

A major objective is to identify software methods for effectively using parallel-processing computers built upon a large number of inexpensive microprocessors. A network of 100 high-performance microprocessors -- each offering a speed of one million instructions per second (MIPS) -- would possess the same raw computational power as a single 100 MIPS supercomputer, but at a far lesser cost. However, that raw power cannot be used to increase the throughput of a single simulation run, unless techniques are developed for discerning at least a 100-fold parallelism in the model.

AMMO is also charged with exporing new simulation technologies for Army applications. Funded research in this area has already endorsed the concept of object-oriented simulation based on a system of behavior inheritances **[Nugent 1983]**, although the particular language evaluated was found to be too slow for current applications.

1.2 Hypercube-Time Warp Research

As noted earlier, AMMO has sponsored research at JPL in the general field of parallel-processing. That research has focused on the issue of effectively using the inexpensive raw computing power that can be obtained by combining a large number of microprocessors. Such use is predicated on exploiting the inherent parallelism available in a simulation.

The JPL team is using a Hypercube system originally developed by Caltech physicists. Based on standard microprocessors, a 32-CPU Hypercube has the raw power of six VAX-11/780's. Of course, this raw power is not the same as the increase in effective throughput, but the actual goal is to maximize the throughput obtained per dollar spent. If the Hypercube only had 15% utilization, or approximately the same power as a VAX but only cost one third as much, then it could be considered to be a qualified success. More importantly, the Hypercube architecture offers extensibility: the design is feasible for 1,000-node or even larger systems.

The software solutions being investigated at JPL are based on the concept of "virtual time" and the Time Warp mechanism of resolving virtual time. The original research in this area was done at the Rand Corporation **[Jefferson 1983]** by David Jefferson (now at the UCLA Department of Computer Science) and Harry Sowizral. Time Warp offers a solution -- perhaps the only one currently available -- towards making effective use of a large number of parallel processors.

The JPL team has implemented a prototype of the Time Warp operating system as a VAX-based simulator in C. It is also implementing the "COMMO*" communications model in C. The group also has plans to do a best-case performance estimate of the model through the critical-path analysis techniques described in **[Berry 1985]**.

Both the Hypercube and Time Warp are discussed in greater detail in Chapter 2.

1.3 CACI

CACI was founded in 1962 to provide instruction in the SIMSCRIPT I language developed at the Rand Corporation **[Markowitz 1963]**. Since that time, the company has been active in the area of military simulation, having developed simulations for all three branches of the armed forces.

Object-Oriented Distributed Simulation

CACI has also worked with Defense Department groups developing their own simulations, both through courses and through consultation in support of SIMSCRIPT I.5 and SIMSCRIPT II.5 compilers. The firm currently maintains simulation compilers across a spectrum of mainframe and microcomputers, as well as more specific simulation tools, such as those for computer and communications networks.

Late in 1984, CACI was hired by JPL to study the requirements for developing a parallel-processing simulation language. This report is a summary of the results of that study.

1.4 Goals of this Study

Based on the requirements of the study's sponsors, the goals of this study were to identify a solution in the following three areas:

1. To increase the ease of maintaining and developing major combat simulations;

2. To support the execution of such simulations in a parallel-processing environment, particularly on the Hypercube under the Time Warp operating system; and

3. To maintain compatibility with existing Army simulation models and modelling teams, if not for syntax, then at least for basic simulation concepts.

The end result of this study is a detailed analysis of the specifications for a simulation language to meet these criteria, as well as the preliminary description of an object-oriented language that fits those specifications.

Chapter 2 describes the available parallel-processing hardware and software systems and concludes with the design objectives for the new language. Chapter 3 summarizes the basic principles of object-oriented programming, including features common to existing object-oriented languages. Chapter 4 describes how those principles are applied in the proposed language.

Chapter 5 describes the use of the language in a parallel environment, with particular emphasis given to distributing the simulation under the Time Warp operating system. Finally, Chapter 6 summarizes the author's conclusions and recommendations.

CHAPTER 2: PARALLEL ARCHITECTURES FOR DISTRIBUTED SIMULATION

2.1 Parallel-Processing Hardware

The popular interest in high speed computing has focused on so-called "supercomputers", such as the Cray X-MP and the CDC Cyber 205. These systems offer the fastest available computational speed for sequential problems. However, their primary speed advantage comes when problems have been expressed as a matrix of related equations; such computation are generally referred to as vector (or array) processing.

For example, the performance of a Cray X-MP-1 has been measured at 21 million floating point operations per second (MFLOPS) when operating as a sequential processor, but increases to 134 MFLOPS when the problem is appropriately vectorized **[Dongarra 1985]**. The improvement of the Fujitsu VP-200 was even more dramatic, from 19 to 220 MFLOPS in the same study. The latter figure represents 150 times a VAX-11/780, the common unit of computation measurement in scientific and engineering computing.

Auxillary array processors are also available for conventional mainframe computers. In either case, this speed can be effectively utilized for problems that can be expressed in matrix form, such as solving large systems of linear equations. The performance improvement will not normally be found when solving problems of a more general algorithmic nautre.

Unfortunately, only a limited class of problems in discrete simulations lend themselves to such vectorized formulations. Line-of-sight calculations, visibility and ranging are examples of areas that could benefit from array processing. But in a typical combat simulation, no single group of calculations occupies a majority or plurality of the computing resources. Existing models are slowed by such general problems as decision tables, intra-unit interactions and event queuing.

This suggests that speed improvements would best be gained by replicating general purpose-computers. This is commonly referred to as a multiple instruction stream, multiple data stream (MIMD) system. If processing and its role in the system is identical, the system is termed homogeneous. Configurations are also possible with one processor acting as a master or control processor and the remaining processors functioning as computational slaves.

One measure of the effectiveness of a paralell-processing system is utilization, or the percentage of the overall computing power (for all processors) spent doing useful work. However, a more meaningful measurement is the overall increase in throughput, or the amount of useful work done by the system. If the individual processor nodes are inexpensive enough, a low-utilization 64-node system may still be more cost-effective than a single-processor mainframe.

The issue of measuring the potential performance of parallel systems is also confused by the wide range of standards available for comparing the performance of the system or its components. The unit of MFLOPS is commonly used in measuring floating point performance, although the number can be used for both single-precision (typically 32-36 bits) and double-precision (60-72 bit) calculations. Performance can also be measured in terms of millions of instructions per second (MIPS), which can be used as a ruler for data accessing and integer calculations.

Where simulated time is maintained as a floating point number -- the most common approach in general-purpose simulation languages -- floating point performance (MFLOPS) is usually the limiting factor in system performance. However, simulations with complex decision logic or elaborate character string manipulations may benefit significantly from an increase in integer performance (MIPS).

2.1.1 Caltech Hypercube

The Hypercube (or "Cosmic Cube," as it is termed in **[Seitz 1985]**) was developed at the California Institute of Technology and is based on a homogeneous MIMD message-passing architecture. A network of 2^N nodes is logically organized in an N-dimensional cube. Each node is connected to N other nodes in an isotropic fashion, and no node is more than N nodes away from any other node. Figure 2-1 shows the toplogy of three-dimensional hypercube.

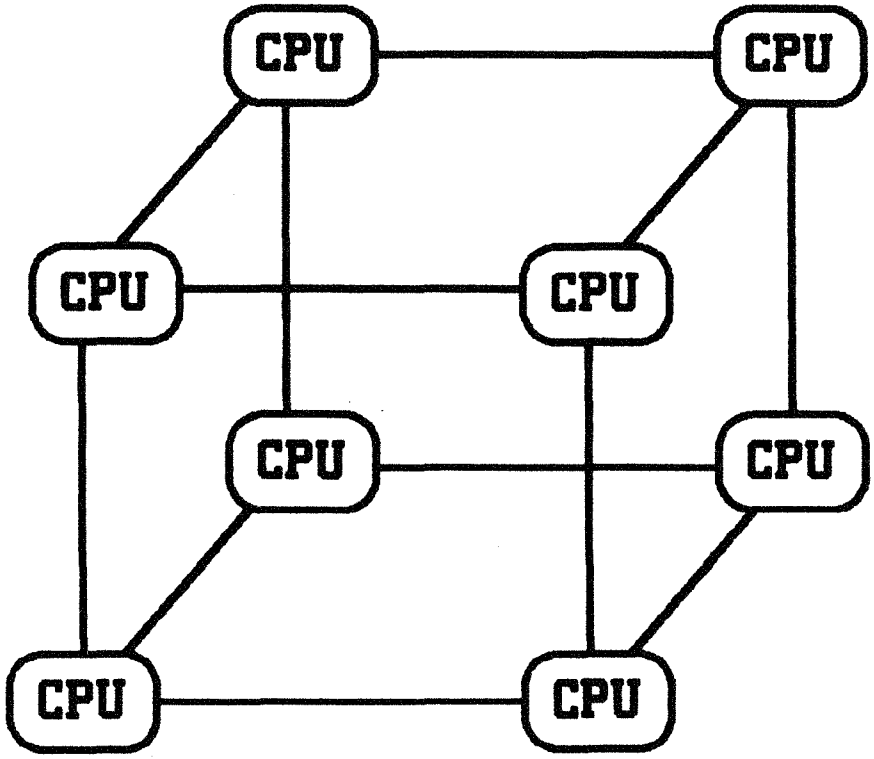


Figure 2-1: Topology of an eight-node Hypercube

Nodes communicate entirely by message passing; no multi-ported or shared memory is available. Communication with each neighbor node is via medium speed (250k bytes/second), bi-directional asynchronous input/output channels.

The Hypercube has two development and control hosts. A VAX-11/780 running VMS contains the software development tools and the intermediate host, which controls the operations of the network. Programs are edited and cross-compiled on the VAX, then downloaded via the intermediate host to one of the Hypercube nodes. The inter-node network is then used to route the program and data to each of the appropriate nodes.

Beginning with a four-node prototype, samples of up of 64 (2^6) nodes have been built. The current version ("Mark II") consists of a cabinet with 32 single-board processors mounted in a rack. Each node has a single-board with roughly the processing power of an IBM PC. An Intel 8086 microprocessor and 8087 floating point co-processor each run at 5 mhz, using 256k bytes of RAM memory and 8kb of ROM. Each node has roughly 20% of the processing power of a VAX-11/780.

Subsequent versions ("Mark III") have been proposed to use a Motorola 68020 CPU as a main processor. The design also includes a second 68020 as an input/output processor, a 68881 numeric co-processor and a minimum of 1 megabyte of memory at each of 32 nodes. Such a node could be expected to be roughly equivalent to 1.2 VAXes,[1] or a six-fold improvement over a comparable Mark II configuration.

With the first prototype completed in early 1982, the Hypercube has been, to date, applied to classical problems of math and physics. It is supported by an applications environment written in C and an inner kernal code in 8086 assembly language.

2.1.2 Intel Personal Supercomputer

The Intel Personal Supercomputer ("iPSC") is another implementation of the Hypercube design. Publicly announced in February 1985, the systems were due to be shipped in the second quarter of 1985.

Each node of the iPSC consists of a board containing the Intel 80286 processor with an 80287 co-processor, and 512kb of RAM. Each node has eight Ethernet communication links, seven of which are used for inter-node communications -- hence limiting the system to 2^7 nodes. User access is through an Intel 310 microsystem, which runs the UNIX-like Xenix operating system on an Intel 80286 processor.

The iPSC systems are configured in cabinets of 32 boards, so four cabinets are required for the maximum 128-node system. The iPSC-d5 is a 32-node system, while the -d6 and -d7 offer 64 and 128 nodes, respectively. System prices are comparable to medium- and large-sized minicomputers. To increase the RAM to up to 4 megabytes per node, every second processor board can be replaced with a memory board. The price of a 16 x 4mb system would be slightly less than a 32 x 512kb system, and so forth.

Unlike the Hypercube, the iPSC offers a global communications channel from the host to the network. Using the eight Ethernet interface chip on each board, this channel can be used for broad-

cast messages to the various nodes, or for intially distributing program and data across the network.

2.1.3 INMOS Transputer

INMOS Ltd. of the United Kingdom has announced plans to develop a series of semiconductor components specifically designed for parallel processing. Because these microcomputers are intended to be used as basic building blocks -- much as happened with transistors in the 1960s -- the company has dubbed its component the "transputer."

The first chip planned is the IMS T424 transputer. A 32-bit microprocessor, it claims a throughput of 10 MIPS **[Wilson 1985]**. Each T424 provides four pairs of high-speed full-duplex data channels. Raw throughput of 10 megabits/second is possible on each channel, although with protocol overhead the effective throughput is approximately 750k bytes/second. Transputer systems are intended to rely on these channels for inter-node communication, without benefit of shared memory.

The T424 is designed for use in distributed systems arranged in two-dimensional grid architectures. In a hypercube configuration, the four-channel restriction would limit it to a 16-node system. More i/o channels are planned in future transputers.

Significantly, INMOS Ltd. does not plan to make machine-language specifications available for transputers, in order to allow architecture implementation changes in future systems. Instead, transputers are directly programmed in Occam, a new medium-level language developed by INMOS for the direct support of parallel processing. (See Section 2.2)

2.1.4 BBN Butterfly

The Butterfly Multiprocessor was first produced in 1981 at Bolt, Beranek and Newman under the sponsorship of the Defense Advanced Research Projects Agency (DARPA). It was originally intended as a packet speech multiplexor for satellite communications, and has primarily been used to implement packet-switching networks.

Most Butterfly systems have been configured as 10 and 16-processor systems, although a 128-processor system has been built **[Goodhue 1985]**. The current implementation has a built-in limit of 256 nodes due to single-byte addresses, although there is no theoretical impediment to building a 1,000-processor machine.

Each processor board consists of a Motorola 68000 processor with 256kb RAM, supported by memory management hardware and an i/o bus. The i/o bus supports both high-speed DMA transfers and the Intel Multibus standard. Additional boards may be added to the node to support up to 4 megabytes of memory or other i/o devices.

The memory of each node is shared throughout the system, and the fundamental communication between processors is through use of the shared memory. The processors independently execute instructions from their local memory, but may also reference memory on a remote processor. It is, of course, implicitly assumed that local references would comprise the majority of all memory accesses.

The nodes are connected through the Butterfly Switch (see Figure 2-2), which combines the techniques of packet switching and sorting networks. The data rate through a single switch path is 32 megabits per second, and the total bandwidth grows "almost" linearly with the number of nodes.

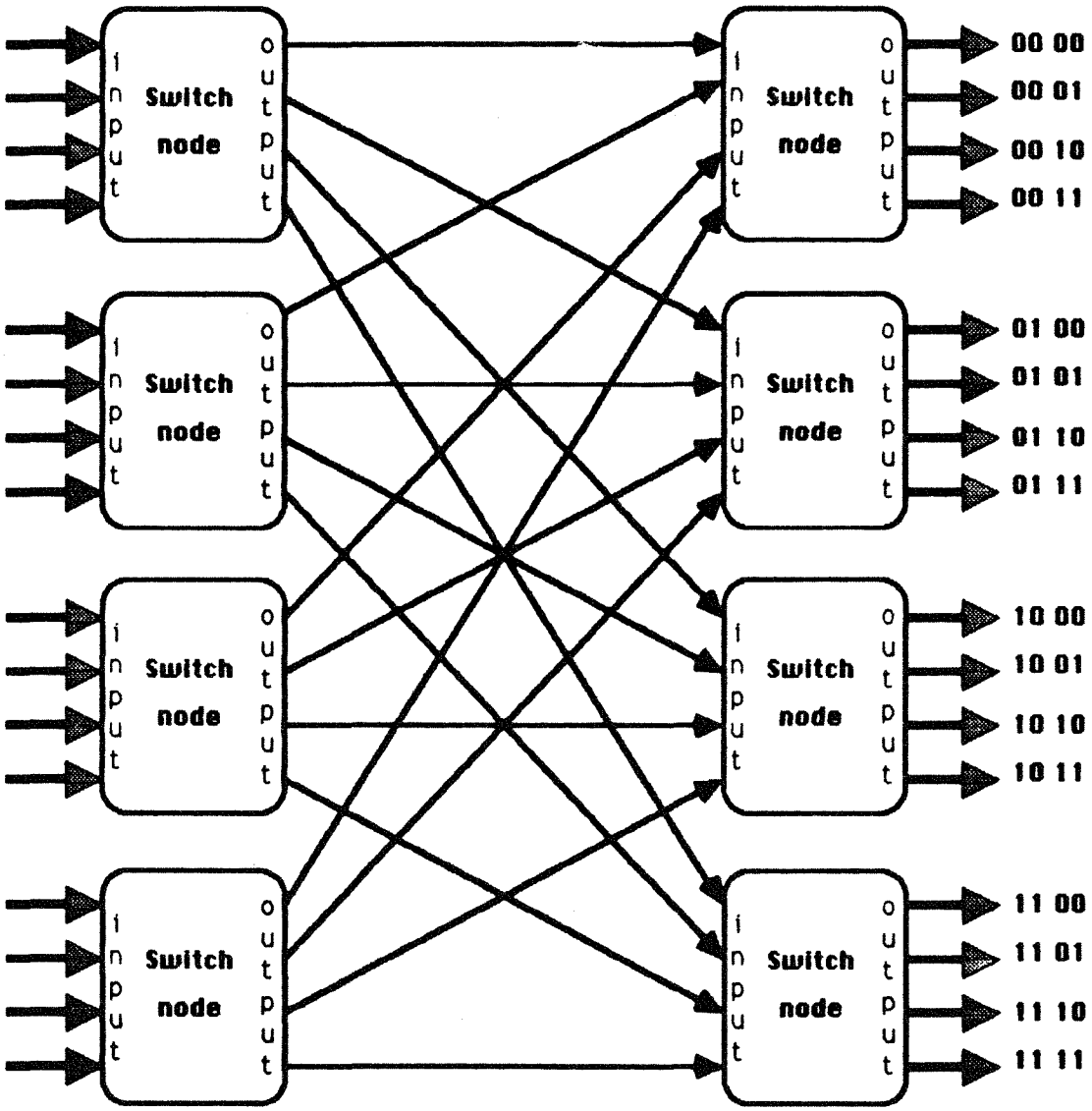
Communication protocols across the switch support:

1. Single-word reads and writes
2. Bulk transfers of data at the full switch bandwidth
3. Various primitive transactions between node controllers, such as queueing, signalling, etc.

The tightly-coupled architecture means the penalty for accessing shared data structures is less severe than with message-passing systems. A remote read of a single 32-bit quantity is five times as slow as a local read, while the ratio for writing is only 3:1.[2] The machine's designers say the Butterfly can use algorithms more similar to those of a single-CPU system, with less attention paid to partioning the problem for a distributed environment that with a conventional message-passing architecture.

Software cross-development is done from a VAX host. An Apple Macintosh personal computer is used as a console terminal and can also be used to boot applications.

The Butterfly is supported by the "Chrysalis" operating system, which is coded in the C programming language, a medium-level language developed to implement the UNIX operating system. Planned system tools include implementation of Ada and Common Lisp cross-compilers.



No matter which line it enters on, a packet will exit at the output line given by its 4-bit binary address

Figure 2-2: The Butterfly Switch

2.1.5 Multi-CPU Mainframes

The DEC VAX-11/782, Gould/SEL PN9080 and Sperry 1100/84 are recent examples of mainframe manufacturers attempting to extend their top-end machine with a small number of identical processors (2, 2, and 4, respectively). Most of the emphasis by the hardware vendors has been on improving the overall throughput of a multi-user system, rather than the coordination of these processors for a single job.

In the case of the Gould and DEC systems, the additional processors are compute-only slaves that cannot perform input/output operations. The master CPU handles interrupts and data transfers for both processors, leaving the slave free to work on compute-intensive problems. All memory is shared between the two processors.

Figure 2-3 shows a diagram of the dual-processor PN9080, which comprises a central processing unit (CPU) and an internal processing unit (IPU). Programs are executed in the IPU until reaching an input or output request. Control of the task is then transferred to the CPU, and the IPU begins executing another task. On the average, the dual-processor configuration records an 80% improvement in throughput over a comparable single-CPU system, from 5 MIPS to 9 MIPS.

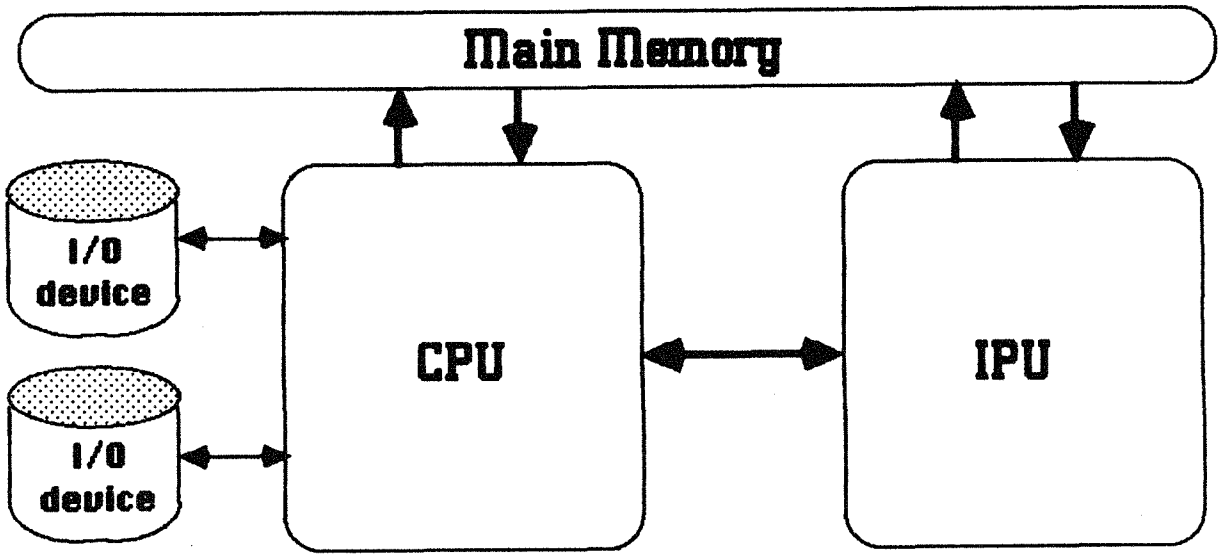


Figure 2-3: A Dual-Processor Gould 9080

The Sperry 1100/80 series allows a number of identical computational processors to be added to the same system; a dual-processor system is referred to as an 1100/82, while the 1100/84 contains four CPU's. In addition, any number of input/output processors can be added to the system; all i/o operations are handled by these processors. All processors -- both CPU's and IOP's -- share common memory. A suspended system task will be transferred to the next available processor of the appropriate type, and thus can sequentially use any or all of the processors in the system.

Even less coupled are independent CPU's that share some or all of the main memory. One example of this is the dual VAX-11/780 system at the SWG lab of the Army Combined Arms Operations Research Activity (CAORA), in which one CPU has been used as a graphics post-processor for a simulation running on the second.

In such higher-level coupled systems, the overhead of the host operating system would tend to interfere with using the system to run a single tightly-coupled distributed simulation. However, research into future configurations along these lines is being actively pursued by mainframe computer manufacturers; future systems may not bear this liability.

2.1.6 Other MIMD Systems

Intel is not the only organization planning to implement a system based on the Hypercube architecture. Inspired by the work of Seitz and the earlier **[Millard 1975]**, Los Alamos National Laboratories is in the process of implementing a 16-node system based on a 10 MHz National Semiconductor 32032 and associated support chips. With two 10 MFLOP floating point processors at each node, the system would offer a raw computing power of 320 MFLOPS. The 16-node configuration is the first phase of a planned, 1,024-node system.

Each node of the LANL system would be supported by 16mb of main memory and a dedicated 512mb hard disk. Inter-node communication would be through as much as 64kb of shared memory for each link. Messages would be transmitted between nodes by using a 20 Mbytes/sec DMA channel to copy data to the shared memory, although some data could "live" in the shared memory and thus in both processors.[3] Classes of applications range from Monte Carlo simulation to artifical intelligence and number theory.

Object-Oriented Distributed Simulation

Another mammoth planned MIMD machine is the New York University "UltraComputer," designed to include a 4096 nodes[**Kozlov 1985**]. Each node would contain a Motorola 68020, floating point coprocessor, memory management unit and several megabytes of memory. The topology of the node connections is the same as for the Butterfly, and the MMU's implement quasi-shared memory, as on the Butterfly. The system is intended to be used for time-stepped simulations with large-scale parallelism, including numerical weather prediction.

A number of systems are planned around the Dataflow concurrent processing architecture, as discussed in Section 2.2.3.

2.2 Parallel-processing Software

For simulation, the most fundamental design point is whether simulation time is identical or different for the various CPU's in the system.

If the simulation uses a principle of synchronous time, then all processors are always synchronized to the same simulated time. This simplifies coordination problems significantly, but is suitable primarily for time-stepped simulations where the symmetry of the physical system is reflected in the computer system -- for example, 16 CPU's simulating 16 identical factory lines. When used with less well-behaved asymmetric parallelism, the time-stepped synchronous approach would tend towards low utilization of many processors.

For this reason, much of the research for distributed simulation has focused on parallel-processing with asynchronous time. In such a system, there is no guarantee that the simulated time on one CPU corresponds to that on another CPU. Processors may race ahead or lag behind in simulated time, depending on their workload. The problem of the parallel-processing system then becomes one of synchronizing activities on the various CPU's where necessary, and assigning the work load to each processor.

2.2.1 Time Warp

Originally developed by Jefferson and Sowizral at the Rand Corporation [**Jefferson 1983**] as a prototype simulator on Xerox Dolphin workstations, the Time Warp principle is currently being implemented by a research group at the Jet Propulsion Laboratory in Pasadena, California.

As the name would suggest, Time Warp uses a relativistic concept of time to coordinate parallel-processing computations. Jefferson uses the term "virtual time" to refer to the asynchronous concept of time in the simulation. As noted earlier, the use of asynchronous time is postulated by many approaches to distributed simulation; Time Warp is distinguished by two additional characteristics:

1. Time Warp relies on implicit synchronization. The user is not required to declare synchronization points. Instead, the Time Warp mechanism works on the assumption that a running simulation is properly synchronized until evidence to the contrary is found. In this way, Time Warp can be seen as taking an "optimistic" view of parallelism, while alternative distributed simulation systems adopt a "pessimistic" view.

2. Because the Time Warp system will eventually find its optimistic assumptions of synchronization not totally correct, a simulation model running under Time Warp will periodically experience a rollback, in which all computations for one or more objects will be undone and retried.

It is the lack of explicit synchronization on the user's part that makes Time Warp an important alternative. The manual synchronization of 3,000 objects occupying 128 processors requires a degree of sophistication likely to be absent in most users. The Time Warp mechanism automatically detects synchronization and corrects cases where synchronization has failed.

Unfortunately, the price for this assistance by the system is a program execution environment that is radically different from that found on a single-CPU system. In addition, without some important information provided by the user's program, simulation language and/or the operating system, many applications would find that the Time Warp overhead could swap any performance benefit gained through concurrency.

The basic framework of Time Warp is an object-oriented system (see Chapter 3) with time-stamped messages communicating between the objects. These objects are referred to as "processes" in **[Jefferson 1985]**. To avoid confusion with the use of that term from a simulation language standpoint, we will instead use the unambiguous term task, which conveys the same meaning in an operating system context.[4]

Object-Oriented Distributed Simulation

Each task has its own clock, which is not usually synchronized with the clock of other tasks in the system. The value of simulated time at each task is referred to as local virtual time. Each inter-task message is stamped with the local virtual time of the sender and the simulated time it is to be processed at by the recipient task.

Simplifying the description somewhat, Time Warp requires that most messages be processed in order of increasing time. As with more conventional discrete event systems, the processing of a message for a particular simulated time changes the LVT of task to be that time.

Receipt of a message out of sequence -- before the task's current virtual time -- causes a rollback to the time of the message. The rollback restores the state of the task by reloading its local data from a saved snapshot.

Once restored, the task re-runs its simulation from that point onward using the new information. All previous incoming messages have been saved, of course, to make it possible to re-execute those messages in addition to the new message causing the rollback.

Meanwhile, the task must undo all the effect it has had on other objects during the cancelled scenario for the given time period. In the current implementation **[Jefferson 1984]**, the Time Warp OS compares the messages generated in the second pass through the period with a list of messages generated the first time through. Again being optimistic, only those messages that do not re-occur must be undone. Jefferson feels that such lazy cancellation is less likely to cause deadlocks, in addition to the obvious reduction in message overhead from the alternative -- which is to cancel all the old messages when the rollback is detected.

If a message is not repeated in the later scenario and must be undone, Time Warp sends an antimessage to the original recipient. The message is identical to the original message in all aspects, except for a "message sign" field, which is negative. If the anti-message and original message arrive at the destination before they can be processed, they mutually "annihilate" each other and disappear from the system. If an antimessage arrives at a task and the original message has already been processed, this antimessage causes a secondary rollback for the recipient task to time of the message.

Progress in the simulation is measured by global virtual time, calculated by Time Warp as the greatest lower bound of the local virtual time of all tasks. Because no task can have an LVT before the global virtual time, it is not possible to send a message with an arrival time earlier than the GVT. This sets a lower bound on the simulated time of rollbacks and, in fact, all housekeeping information (saved snapshots, message lists, etc.) from earlier than the GVT is discarded.

However, any action in the simulation later than the global virtual time is subject to rollback and must be considered tentative in nature. Input/output tasks therefore must queue all data output to external peripherals until the GVT exceeds the corresponding data's time stamp. For diagnostic purposes, however, it might prove valuable to show graphics displays for Time Warp tasks as the output messages are received, and erase the display and start over when rollbacks for the graphics task are necessitated.

2.2.2 Chandy-Misra

As noted by **[Jefferson 1983]**, there are a number of proposals for distributed simulation that adopt a network view of a discrete simulation. The simulation is decomposed into so many nodes (objects) which are linked along well-defined paths corresponding to the inter-node interactions.

As with Time Warp, these approaches view time as asynchronous. Each object has its own local simulation time, and messages are time-stamped as to their desired arrival time.

However, a strict order is imposed on the message arrival. Each path of the network is assumed to contain sequential messages with non-decreasing arrival times. Although any number of input path may exist, once a simulated time is reached on one path, there is no turning back to the earlier time. A node can examine the first message queued on each path, and then take the oldest of the messages and advance time to that point, since there is now no way for an older message to be received. The most conservative approach won't allow a node to continue unless there is at least one message in each input queue.

Object-Oriented Distributed Simulation

This works well in a simple hierarchical or sequential topologies, as shown in Figure 2-4. A series of a messages can travel in one direction along the network, and because there is no cycle in the network graph, there is no possibility of deadlock in the system. However, if the **B** nodes are faster at generating messages than the **C** node is at processing them, the approach is subject to potential memory overflow as messages pile up on **C**'s input message queue.

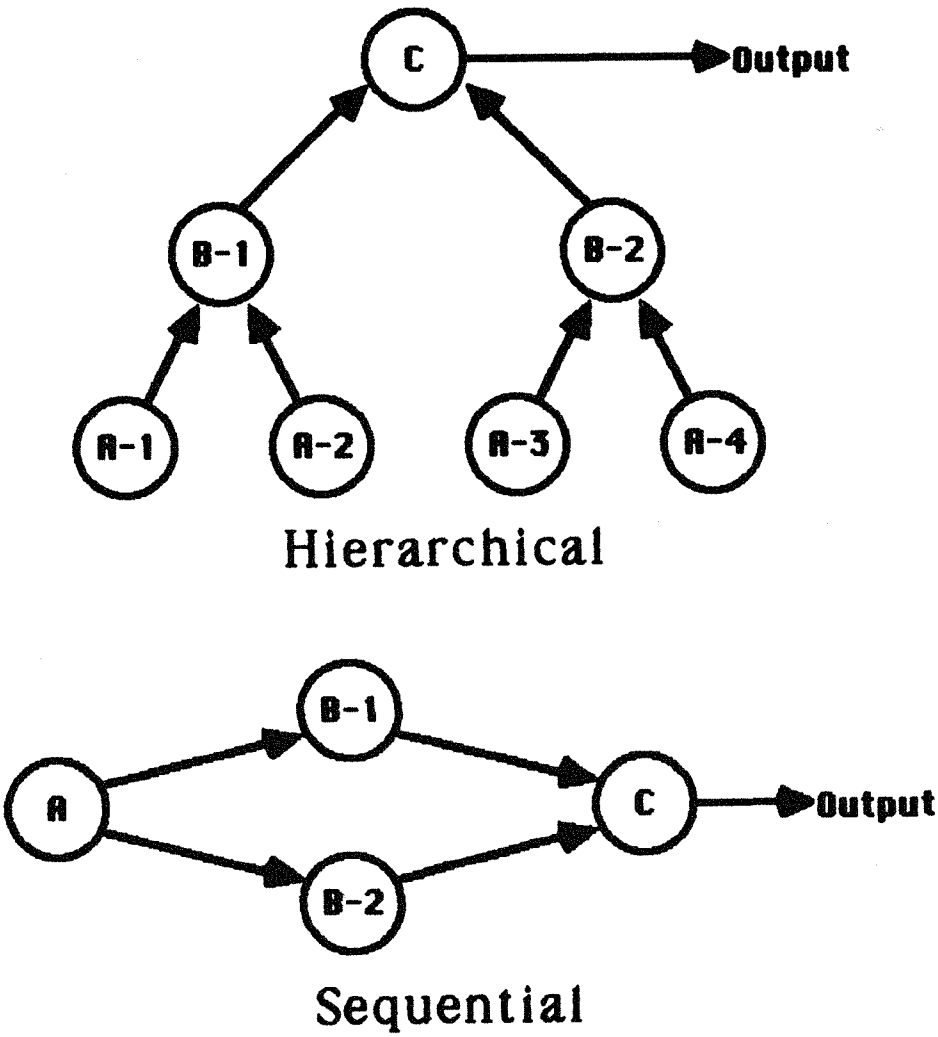


Figure 2-4: Simple Networks for Distributed Simulations

These simplified topologies cannot be said to representative of the broader class of simulation problems. All but the most extreme of problems contain at least one cycle in the system, as in the modified topology of Figure 2-5. A conservative approach is guaranteed to deadlock, since no node can continue without at least one message on each input queue.

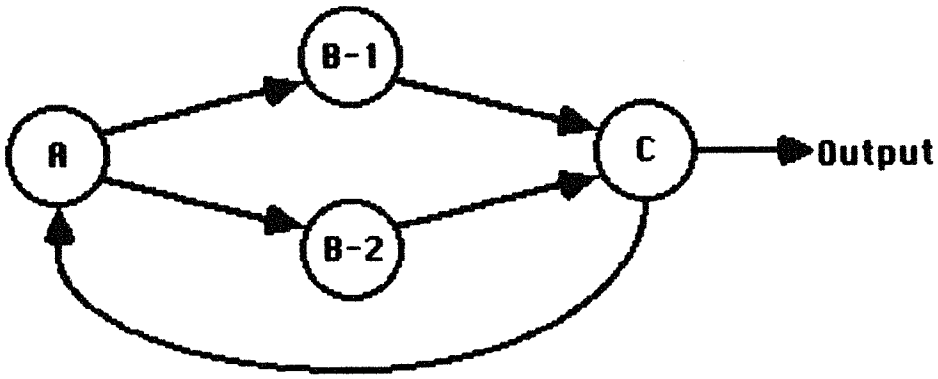


Figure 2-5: Simple Network with a Cycle

As noted by **[Jefferson 1983]**,

> ...the likelihood seems remote that a large, complex simulation will succeed, at every node and around every cycle, in walking the narrow path between these two dangers of deadlock and memory overflow long enough to terminate normally.

K. M. Chandy and J. Misra of the University of Texas have proposed several additions to the network concept in various papers, including **[Chandy 1981]**. Message queues along each path have a finite length under the Chandy-Misra proposal, and not only does an empty queue stop a node on input, but a full queue stops a node's output.[5]

This modification allows a finite limit on the number of messages in the system and thus on the total memory requirements. However, the limit on computation output exacerbates the problem of deadlocks, since even less computation is being done than before.

To alleviate this problem, a deadlock-detection mechanism is used to automatically find and break deadlocks. When a deadlock is discovered, each node sends a null message with its current simulation time. The null messages can be used to calculate a concept comparable to Jefferson's global virtual time, and then at least one task can be restarted without worrying about the possibility of an earlier message coming in.

A number of other approaches to distributed simulation networks simulation exist, including **[Ziegler 1985]**.

2.2.3 Dataflow Languages

A number of researchers have concentrated their parallel-processing research efforts on examining the sequential relationships of data used in complex calculations. The class of solutions that have resulted are collectively referred to as dataflow. Although not directly related to discrete event simulation, there are a number of parallels to the previous approaches, as well as potential solutions to the general problems subsumed by distributed simulations.

The dataflow approach structures a problem as a hierarchy of directed graphs establishing the causal data relationship. The graphs are typically coded for **small grain** dataflow, which assumes instruction-by-instruction parallelism. Such low-level parallel processing is comparable to the instruction pipelining of large mainframe computers, but is much more sophisticated and extensive in scope.

Such algorithmic parallelism can be expressed in so-called "single assignment" languages, such as SISAL **[McGraw 1983]**. In such languages, each variable can be assigned only once, but used multiple times. The following SISAL program integrates the area under the curve $y=x^2$ in the interval **[0,1]**:

```
function integrate ( returns real)

for initial
     int := 0.0;
     y   := 0.0;
     x   := 0.02;
while
     x < 1.0
repeat
     int := 0.01 * (old y + y);
     y   := old x * old x;
     x   := old x + 0.02;
returns
     value of sum int
end for

end function
```

The loop can be unwound and executed in parallel, similar to the Occam **par** construct. However, the **old** construct is used to establish the sequential relationship between successive iterations of the loop, thus restricting an iteration from using an **old** value before it is set by the previous iteration. The single-assignment constraint means that any graph node waiting on a data value can begin execution once the value is set, since it cannot later be reset. At the conclusion of the loop, the sum of the parallel calculations is available and is returned.

The best known small grain dataflow machine is the Manchester Dataflow Processor, described in **[Gurd 1985]**, from which the preceding example was taken. The system uses a tagged token memory architecture to represent the data values of the graph. In a benchmark of 1-12 function units, system efficiencies range from 100% to 91%, yielding an 11-fold throughput improvement for the 12-processor case.

Research into **large grain** dataflow is less frequent. With parallelism (and dataflow graphs) specified at the routine level, the graph of a large grain dataflow program is equivalent to the network simulation paradigm of Chandy and others. A large grain dataflow system exchanges the greater processor utilization of the small grain approach for a decrease in communication overhead.

2.2.4 Occam

Although it lacks direct support for simulation, the recent design of the language "occam" **[Inmos 1984]** offers a number of important ideas on how a language can provide user support for parallel processing. It adopts a quasi-dataflow approach to small-grain parallelism, waiving the single-assignment restriction of SISAL while requiring explicit declaration of task interactions.

Occam uses data channels for communication between parallel tasks, which function roughly as single-variable pipes**[May 1984]**. The two channel input and outputs operators are a fundamental building block of an occam task:

chan ! value outputs **value** to channel **chan**
chan ? value inputs **value** from channel **chan**

A series of tasks may be executed using one of the following three replicating constructs:

SEQ sequential execution of tasks
ALT the first task ready is executed
PAR all tasks are executed in parallel

For example, a simple vector addition could be performed as

```
PAR i = [0 FOR n]
   a[i] := b[i] + c[i]
```

Indentation, incidentally, is not a matter of programmer preference, but a required syntactic specification of block structure.

A more complex example from **[Wilson 1984]** is the evaluation of a common signal processing problem, the butterfly fast Fourier transform expressed by the equation of complex variables:

$$X = A+B$$
$$Y = W(A-B)$$

If each butterfly of the FFT is allocated to a single transputer, this is expressed in occam as

```
WHILE TRUE
  SEQ
    PAR
      A ? Areal;Aimag
      B ? Breal;Bimag
    PAR
      X ! Areal+Breal;Aimag+Bimag
      Y ! ((Areal-Breal)*Wreal)-
            (Aimag-Breal)*Wimag);
          ((Aimag-Breal)*Wimag)-
            ((Aimag-Bimag)*Wreal)
```

The algorithm must be modified if each transputer is to perform multiple butterfly calculations.

2.3 Design objectives

From the common characteristics of the parallel processing systems outlined earlier in this chapter, a number of objectives were postulated for the design of an approach to distributed simulation. The objectives, it is hoped, reflect the goals of this study, as outlined in the previous chapter.

The following seven points represent the ideals by which the proposed concurrent simulation language should be evaluated:

1. **Support object-oriented programming based on a specification of inherited properties.**

   The current technology of many-node parallel processors emphasizes the use of message-passing for inter-processor communication. Such a constraint maximizes the difficulty in exploiting parallelism in a program's execution, and so represents the technical obstacle that should be overcome first. A solution for a message-passing architecture will also work in an environment of shared memory, or on a system of nearly-shared memory, such as the Butterfly.

   In addition, the Time Warp concept relies implicitly on state changes through (time-stamped) object-object messages. It is the only mechanism provided for communication and synchronization between tasks running on different CPU's.

While the object-message paradigm alone is a powerful one, the greatest productivity benefits are promised in conjunction with behavior and attribute inheritance. For example, Apple Computer has plans to develop a library of inherited default behaviors as a way of building programs for its Macintosh personal computer.[6] Such an approach holds great promise for simulation. The strength of the existing SIMSCRIPT system is based on a wealth of standard behaviors generated for certain classes of objects, although there is limited flexibility in enhancing those behaviors.

2. Support stuctured programming, where not inconsistent with (1).

In the past 15 years, modern programming language development has focused on the problem of developing and maintaining complex software systems using teams of programmers. Such work has led to what is conventionally termed "stuctured programming." Such languages offer a number of advantages for large programming projects that should be extend to the area of object-oriented programming, where possible.

The unit concept of UCSD Pascal **[Clark 1982]** was perhaps the earliest approach to the problem of separate compilation and development. A group of procedures is clustered in a unit; outside the unit, only specifically-defined interfaces are available. This separates the external specifications of a unit from its internal implementation. This concept was further developed in Modula-2 modules and the package concept of Ada.

However, some of the ideas of structured programming are antithetic to true object-oriented programming. Strong compile-time checking -- even in conjunction with Ada's identifier overloading -- does not allow for the flexibility of a run-time determination of an object's state variables or behavior. In addition, many of the principles of structured programming only further encourage the use of global or quasi-global data for communication between procedures. This makes even more intractable the task of partioning data and computations for parallel processing, as will be discussed in later chapters.

(3) Allow efficient implementation on numeric processors.

While "Lisp machines" and other associative processors hold great promise for the future, in the near term, mass-production microprocessors will emphasize conventional numeric architectures. For example, each of the parallel-processing systems cited earlier in this chapter is based on standard register-machine architecture.

In the meantime, the use of Lisp or similar languages on conventional processors commands a performance price too high to meet the AMIP objectives described in Chapter 1. In one study using the Lisp-based ROSS **[Nugent 1983]**, the simulation was estimated to be 10 times slower than required for a production model. Even using the Lisp Flavor System and the most efficient numeric processor implementation available, reference **[Elias 1985]** estimated that a Lisp-based simulation was four times as slow as one written in a procedural language.

However, many object-oriented productivity gains are obtainable on conventional processors by using a fully-compiled language. Both C++ **[Stroustrup 1984a]** and Object Pascal **[Tesler 1985a]** suggest that the fundamental objectives of **(1)** and **(2)** can be met without compromising performance. In fact, the former report claims slightly better performance for C++ over less compact formulations in C, in some cases.

The experience of **[Cosell 1984]** is perhaps a more realistic predictor, where a multiple-path inheritance system brought a measurable, but acceptable performance degradation. Certainly computational resources are not so inexpensive as yet to be able to accept even a twofold slowdown without serious qualms.

(4) Directly support discrete simulation.

As noted by **[Law 1982]**, a simulation language offers a number of significant advantages over a general-purpose language such as FORTRAN or Ada:

A. Simulation languages automatically provide the tools needed for writing simulations, increasing programmer productivity.
B. They provide a natural framework for describing and implementing models.
C. Models become shorter and easier to maintain.

D. More compact programs are less likely to have errors.
E. Automatic compile-time and run-time error checking is provided for common simulation problems.

The disadvantages noted by Law are that specialized tools require additional training, and a user may not always find those tools on a particular host computer. For major simulation efforts, this additional cost is generally far smaller than the savings provided by these advantages over the lifetime of the model.

To ignore the two decades of development and modeling using the appropriate tools would be a giant step backwards, in the author's opinion. The failure to provide appropriate simulation tools -- no matter how good the language might be -- would increase the risk of user rejection and failure.

(5) Inclusion of second-generation simulation constructs.

As noted in **(4)**, first-generation simulation languages provide improved user productivity over the direct use of general purpose languages (typically FORTRAN) for discrete-event simulation.

Later languages have taken these tools a step further, by providing a more natural framework for the description of the steps in a discrete simulation. An example of this is the use of processes to group a series of related action by a single actor in the simulation.

For example, the following pseudo-code illustrates such an association:

```
process TRUCK
     request a GAS_PUMP
     wait 5 minutes        ''pumping
     relinquish GAS_PUMP
     obtain ROUTE_ASSIGNMENT
     request DRIVER
     work 2 hours
     relinquish DRIVER
end
```

Alternatives to the process view require formulation of many simpler events, often along an artificial division that does not correspond to a fundamental conceptual flow of the problem. In contrast, processes cluster groups of related actions into a common procedure, a primary goal when building a behavior-oriented simulation.

As implemented in SIMSCRIPT II.5 **[Russell 1983]**, processes have formed the basis of a number of major military models, such as CASTFOREM, JTLS and SCSS. The process approach is also an integral part of Simula, as described by **[Birtwistle 1984b]**.

The specification of sequential actions for a given object is also reflected in the activity-block structure of GPSS, although GPSS lacks the flexibility of a general-purpose high-order language that is necessary for implementing complex simulations.

A third-generation simulation language should also improve upon previous implementations of other concepts, such as the handling of resources in SIMSCRIPT II.5 and Smalltalk-80.

(6) Appropriate for single and N-processor configurations.

A concurrent simulation language should give the user a conceptual framework that provides clues to parallelism that help divide up the computation and data. At the same time, the design should not penalize use of a single-processor system.

Of course, any program that runs on "many" processors could run on one, with a simulated multi-processor network, if necessary. But much of the overhead associated with interprocessor communication and synchronization would be unnecessary and could be accomplished through more direct means. For example, sending a message could be mapped into a direct call of the appropriate method routine.

An important requirement is that moving between a single and multi-processor configuration be virtually transparent to the user. This also applies when the parallelism of the simulation increases (more objects) or decreases (fewer objects).

Failure to provide for such portability will bring on impractically complex problems of configuration control. Most large models will move between a variety of hardware configurations within their lifetimes -- or even within the development period; for example, with short runs on a VAX and longer runs on a Hypercube. Spanning both extremes also guarantees support for MIMD machines with characteristics somewhere in between, such as the quasi-shared memory architecture of the Butterfly.

(7) Interface to the Time Warp operating system.

Despite its use of a "brute force" approach to resolving synchronization problems, the Time Warp operating system currently appears to be the only approach near enough to fruition for use in distributed simulation. In addition, the highly structured formalism of the alternatives to Time Warp imply greater redesign efforts (in addition to recoding) to move existing simulation models to distributed systems.

In accepting Time Warp, one must also accept a number of additional requirements for a simulation model and language. These are usually in the form of additional information required by the operating system, or restrictions on user programs. An example of the former is time-stamping messages; an example of the latter is the prohibition on direct global memory accesses. These might or might not be relevant to another approach to distributed simulation but could, at worst, be ignored by the compiler and operating system when running on another system.

Among the programming languages surveyed as a part of this study, none meet all seven criteria: most pass only two or three. The description proposed in Chapter 4 is an example of a language that meets all seven of these objectives.

CHAPTER 3: BASICS OF OBJECT-ORIENTED PROGRAMMING

This section explains the fundamental concepts of object-oriented programming, as well as a number of implementation alternatives. A comparative analysis of existing object-oriented languages is beyond the scope of this chapter, but is covered in Appendix B.

3.1 The Object-oriented Programming Concept

Object-oriented Programming is a paradigm for computational processes; that is, a model of how computation is performed by a machine as seen by the programmer. There are a number of such paradigms, each leading to a different view of what a computer is and how it behaves.

A "register machine" model is typified by a classical assembly language program. "Functional programming" is the essence and was the original motivation for Lisp. The "Logic programming" paradigm is commonly associated with the language Prolog, but implicit in most database query applications, such as dBaseII. Reference **[Abelson 85]** is a basic textbook on the structure and interpretation of computer programs, and includes a comparative study of various paradigms within the framework of a single pedagogic language.

There are two uses for these paradigms. First, they can serve as the basis for the construction and use of computing machines at the software level. Second, they can be used as inspiration for programming styles in any language or machine. Thus, we find the object-oriented concept both in languages, such as Smalltalk-80, and applications written "in the object-oriented style" in other languages such as MIT's Lipsim Air Traffic Control simulation.

These paradigms are rarely found in their pure form. For instance, Lisp is a functional programming language with some reluctant concessions to the assignment model. This is why we refer to closed and open implementations of object-oriented programming. In the former, the programming environment strictly enforces the paradigm to the point that no other programming method is available. In the latter, its use is possible, facilitated, or even strongly encouraged, but alternate forms are available, even if discouraged. Thus, we would say that Lisp is an open implementation of functional programming, while Smalltalk-80 takes a closed object-oriented approach.

3.2 Objects Have Local State And Functionality

Object-oriented programming views programs as being built around conceptual entities that can be likened to real-world things. Each of these entities, called objects, has a set of operations that can be performed on it. The first design step in creating an object-oriented program is to determine the objects that are going to exist. For example, in an Air Traffic Control (ATC) simulation, the objects may include aircraft (which carry altimeters, airspeed indicators, heading indicators and other instruments) radars, communication links, displays and display images, etc.

This first design step forces the programmer to think of the real objects that are going to be modeled and to establish a one-to-one correspondence between these and the computer artifacts which the program will manipulate. This makes the program logic much more transparent and easy to follow since it closely resembles the way people organize their knowledge of a system.

Like their real-world counterparts, objects can be grouped into classes (or types), so that each member of a class exhibits similar behavior. Indeed the aircraft, altimeters, etc. mentioned above did not describe specific objects, but classes of similar objects. An object-oriented program therefore defines a number of object types, a set of operations allowable for each object and can create a number of instances of each type. For example, an object class **AIRCRAFT** may be defined, then three instances (three actual objects) of type **AIRCRAFT** may be created and manipulated by the program. One might have the name "TWA 611," a second named "PAN AM 7" and a third "United 436."

In order to distinguish two instances of the same object type, each object must maintain its own internal state information. A number of terms are used to describe an object's internal state: state variables, attributes, slots, instance variables, are but a few. We will use the term attribute to conform with the established SIMSCRIPT terminology. An object's attributes can be examined and altered using the operations that are defined for this object class.

The class **AIRCRAFT** may have attributes which include its current position (**LATITUDE, LONGITUDE, ALTITUDE**), its current speed vector, (**NORTH.SPEED, EAST.SPEED, VERTICAL.SPEED**) all the onboard instruments, etc.

Some of the operations that can be performed on aircraft might involve simply accessing the appropriate attribute, such as **GET.AIRCRAFT.LATITUDE, GET.AIRCRAFT.LONGITUDE, GET.AIRCRAFT.ALTITUDE**, etc. In addition, we can define operations like **SET.AIRCRAFT.LATITUDE**, to alter those attributes. Finally, we can define operations such as **GET.AIRCRAFT.SPEED, GET.AIRCRAFT.DIRECTION** which, rather than simply returning the value of an attribute, perform the necessary operations required to return.

Even in this very basic form, object-oriented programming style helps and encourages the design of simple, modular programs. Since the state of any object can only be manipulated directly by a well defined set of operations, these become the natural interface of the object with the rest of the world. The object becomes a black box which behaves in a well-defined way, while the remainder of the program is not required to have any knowledge of the internal logic and structure of the object. This is consistent with the principles of structured programming that began with Algol-60 and are now typified by Pascal, Ada and Modula-2.

We note here that SIMSCRIPT II.5 already includes many of the notions of object-oriented programming according to the description so far. In particular, when a SIMSCRIPT program defines a temporary entity it is creating a new object class:

```
temporary entities
     every AIRCRAFT has LATITUDE, LONGITUDE, ALTITUDE,
          NORTH.SPEED, EAST.SPEED, and VERTICAL SPEED
```

In addition, SIMSCRIPT II.5 automatically defines a number of operations on **AIRCRAFT.** For example, **CREATE** an **AIRCRAFT** called **AC1**, would make an instance of an aircraft; **LATITUDE(AC1)** returns the value of the aircrafts latitude, etc. Finally, the user is allowed to define his own operations on **AIRCRAFT.** This fact is recognized in section 4, where we propose to use the existing temporary entity concept as the starting point for all enhancements toward, a fully object-oriented extension of SIMSCRIPT II.5.

Another characteristic of objects is individuality. Consider two instances of **AIRCRAFT**, **AC1** and **AC2**. These two aircraft may have exactly the same state variables. However, they are not the same aircraft; an object's individuality is distinct from its state description. If **AC1** refers to the same instance (memory block) as **AC2**, then we say that **AC1** is equal to **AC2**. If **AC1** and **AC2** are not equal, but the values of all their attributes are identical, we state that **AC1** is equivalent to **AC2**. Note that in conventional numerics programming, the identity question does not arise, so that there is no dichotomy between the terms equal (i.e., being the same object) and equivalent (i.e., having the same value).

To pursue this a bit further let us consider another object type named **BOX** defined as:

temporary entities
every BOX has LENGTH, WIDTH, HEIGHT,
THICKNESS, WEIGHT, and CONTENTS

Let us also assume that for some implementation, the attributes of an object are represented by one dimensional arrays so that if **BOX1** is an instance of **BOX**, **LENGTH(BOX1)** would reference the first element etc. It is obvious that **BOX**es would look exactly like **AIRCRAFT** at runtime; there would be no way to answer the question "Is this array a representation of an object or is it an ordinary array?"

We refer to these variables as statically typed variables. This means that implicitly or explicitly we have to tell the program a priori what type of object this variable represents (points to). Static typing (or strong typing) is a contract between the programmer and the compiler (or interpreter) which states that each variable can point to one and only one type of object.

The opposite practice is called dynamic typing. A dynamically typed symbol is a pointer to an object whose type can be determined at runtime. The information on what type of object it is may reside in the object or in the variable itself. In the first case, the object's state contains information on the type of object it represents. In the second case, the "address" of the value object includes the type information, which is equivalent to extending the machine's address space n-fold for n objects.[7]

In both static typing and dynamic typing, the object's type is checked at compile time or run time, respectively.

The third alternative is untyped language, in which the type is

never checked. Obviously, this provides the maximum flexibility and the greatest risk of an undetected logical error.

The consequences of strong typing are far-reaching. In our example above any operations defined on **BOX**es will work with no run-time errors when given an aircraft as an argument and vice-versa. In such a system, the burden of insuring that each operation is performed on the appropriate object is placed squarely on the shoulders of the user. On the other hand, dynamic typing imposes a runtime overhead since all operations on objects will be required to do type checking on their arguments.

3.3 Generic Operations on Objects

Let us consider a simulation of a radar tracking aircraft targets. The program needs to know where each aircraft is and will therefore use **AIRCRAFT.LATITUDE**, **AIRCRAFT.LONGITUDE** and other operations available for aircraft objects. However, if the radar is next required to track missiles, it would have to use the equivalent operations for missiles, since **AIRCRAFT.LATITUDE**, **AIRCRAFT.LONGITUDE**, etc. would not work on missiles. This means that we will need two similar segments of code, one referencing missiles and another referencing aircrafts.

A single-code segment could be made to track both aircraft and missile targets if we have a generic operation, such as **GET.LATITUDE**, which when invoked for an object returns the object's latitude. In this context, **GET.LATITUDE** is not a particular function but rather the name of an operation. The Smalltalk phrase "sending a message" is commonly used to denote a request for the performance of a generic operation on an object.

In order to perform this operation we need to know the object's type and the name of the operation to be performed: for each defined object type, the system keeps associations between the name of the operation and the actual function to be invoked, called a method. Thus, all **AIRCRAFT** objects share a **GET.LATITUDE** method, a **GET.LONGITUDE** method, etc., and these are distinct from the **GET.LATITUDE** and **GET.LONGITUDE** methods for objects of type **MISSILE.**

In summary, we have the following universal protocol for performing a generic operation on some object: we send the object a message consisting of the name of an operation (e.g. **GET.LATITUDE**) and possibly some arguments. The actual method executed may return a value, or may perform a side-effecting operations, but in any case the effects of the message can depend on the type of object which receives it.

The following should be noted about message-passing:

1. If we perform an action on an object, this implies that objects were passive elements of the program the caller determines how to operate on the object. With generic operations and message passing, the object itself determines which function is invoked and therefore the exact effects of the message. From the user's standpoint, the object is active; it receives messages and it responds either by returning a value, by some side effect, or both.

2. The concept of generic operations is not new. The need for generic arithmetic functions was identified and implemented since the early days of FORTRAN. What is new is the mechanism by which the user defines and uses generic operations. This simple mechanism has become one of the most powerful concepts of Object-Oriented programming.

3. Even though knowledge of the object's type is required for correct fielding of messages, dynamic typing is not a necessary prerequisite for generic operations. Indeed, arithmetic functions in compiled languages like FORTRAN, PL/1 and SIMSCRIPT (all of which use strong typing) are generic operations.

3.4 Inheritance Of Attributes And Behavior

Let us consider an object of type **TANK.** Like an **AIRCRAFT**, a **TANK** has attributes **LATITUDE, LONGITUDE, ALTITUDE, NORTH.SPEED, EAST.SPEED** and **VERTICAL.SPEED.** Tanks should accept messages like **SPEED** and **DIRECTION** just as aircraft do. This could be achieved by including in the definition of **TANK** all the attributes and methods of **AIRCRAFT** that pertain to horizontal movement. However, this is both inefficient and undesirable since we would have to maintain two copies of essentially the same code.

Alternatively, a statement could be made that **TANKs** and **AIRCRAFT** are similar with respect to the operations **SPEED** and **DIRECTION.** One way of stating this is to declare that both **TANK** and **AIRCRAFT** are subclasses of a more general class of object which embodies the behavior common to both **TANKs** and **AIRCRAFT.**

We will first build a simple and more general object which we can call **MOVING.OBJECT.** Then, we state that **AIRCRAFT** behaves like a **MOVING.OBJECT** -- but can also do other things, such as fly -- and

that **TANK** also behaves like a **MOVING.OBJECT.** This could be accomplished by a syntax such as:

> **every LOCATION has a LATITUDE, a LONGITUDE,**
> **and an ALTITUDE**
> **every MOVING.OBJECT is a LOCATION and**
> **has a NORTH.SPEED, EAST.SPEED, VERTICAL SPEED,**
> **a NORTH.ACCELERATION, EAST.ACCELERATION,**
> **and a VERTICAL.ACCELERATION**
>
> **every AIRCRAFT is a MOVING.OBJECT and**
> **has a BANK.ANGLE and a LONGITUDINAL.ACCELERATION**
> **every TANK is a MOVING.OBJECT and has a GUN and a CREW**

An object class **LOCATION** is defined with three attributes. In the next definition, class **MOVING.OBJECT** is given six attributes, but at the same it is stated that is a **LOCATION.** We would like this to mean that the class **MOVING.OBJECT** inherits the attributes of class **LOCATION** so that it has nine attributes.

Furthermore, we would like **MOVING.OBJECTS** to inherit all of the behaviors of **LOCATION.** If, for example, objects of type **LOCATION** return the object's latitude in "dd:mm:ss" format when they receive a **LAT.DMS** message, we would like objects of type **MOVING.OBJECT** to act the same way.

The final two definitions specify that **AIRCRAFT** and **TANKS** behave like **MOVING.OBJECTS** except that they have some additional attributes (and possibly will be capable of receiving additional messages). We have thus built a hierarchy of objects, starting from a simple and generic one (**LOCATION**) and ending with more complex and more specific ones (**AIRCRAFT** and **TANK**).

There are different opinions regarding which object should be considered to be "above" which one. We use here the nomenclature of Smalltalk-80, which calls the more primitive object (e.g. **LOCATION**), the <u>superobject</u> and the more complex one, (e.g. **AIRCRAFT**) the <u>subobject</u> thus, **MOVING.OBJECT** is a <u>superclass</u> of **AIRPLANE**, and a <u>subclass</u> of **LOCATION.**

An important alternative in the implementation of object-oriented systems is whether to limit the hierarchy of classes to simple trees, or to allow arbitrary directed graphs.[8] In the second case, a class may have more than one superclass, and rules may be necessary to resolve possible inheritance conflicts (e.g. two superclasses have identically named attributes). On the other hand, multiple inheritance has been found to be of practical value in coding actual object-oriented simulations.

Object-Oriented Distributed Simulation

In creating a new class of objects, there are situations when the behavior of some existing object is almost, but not quite, what is needed as a basic building block; it would be desirable to inherit all but a few of the superclass attributes or operations. This can be accomplished by defining attributes or methods in the subclass with the same name as the superclass attribute or method we want to block. This concept is called overriding (in the case of simple tree class hierarchy) or shadowing (in the case of multiple inheritance).

Going back to our aircraft example, we notice that moving objects have methods to return their north and east accelerations. At the same time, aircraft have bank angle and longitudinal acceleration as attributes. This means that the aircraft's attributes are overspecified, since north and east acceleration can be determined by the bank angle and longitudinal acceleration.

The problem can be resolved by defining an operation on aircraft called **GET.NORTH.ACCELERATON** which, instead of interrogating the **NORTH.ACCELERATION** attribute, computes the appropriate value from the aircraft's bank angle, longitudinal acceleration and current direction of motion. Whenever an aircraft receives the message **GET.NORTH.ACCELERATION** it is this method, and not the one defined for **MOVING.OBJECT**, which will be invoked. The new method "shadows" the one provided by MOVING.OBJECT so that the latter is not even visible from the user's point of view.

This example raises an important implementation issue: how are attributes accessed from inside a class's method? Three alternatives are available:

1. Direct access, that is, not involving any "secondary" message sending.

2. Exclusively by means of a message, possibly created automatically by the system (to avoid the recursive definition problem).[9]

3. Allow the user both types of access.

At first glance, direct access is the more efficient alternative, since message passing will always be more expensive than simple memory reference. On the other hand, many of the advantages of shadowing, (as illustrated in the previous example with the **AIRCRAFT NORTH.ACCELERATION)**, are lost since the user must be very careful about "real" vs. "virtual" (i.e., method-implemented) object attributes.

Smalltalk-80 chooses the second alternative (preventing the user, by the way, from overriding the system-defined methods), while ROSS and the Lisp Flavor System choose the third alternative.

On the balance, one would conclude:

1. The user must be allowed to override the attribute-access methods (therefore, we must choose alternative 3); on the other hand,

2. Direct access should be used as sparingly as possible, for example only in the attribute-access methods.

A related issue is whether to allow direct access to object attributes from outside the object's methods (e.g., from an unrelated class method). This is exactly the current SIMSCRIPT II.5 temporary entity attribute access mechanism: anybody knowing the identity of an object can directly access that entity's attributes. This conflicts with the basic rule of Object-oriented programming:

> Advertised operations on any object class are the established interface between the object and the outside world. As such, they are subject to shadowing or other modification. It is, therefore, an error to bypass the message-passing mechanism when accessing an objects attributes unless strictly necessary.

It should be noted also that direct access of attributes outside a method are inconsistent with the object-message architecture of the Hypercube and similar systems, and the concept of virtual time under the Time Warp operating system.

Methods for an operation that are totally replaced by another method when shadowed or overriden are called the <u>primary method</u> for that operation. Methods however, can be attached to a method inherited from a superclass for a given operation without replacing it. This is useful to enhance the existing method by adding some extra processing, change the default values of arguments to the methods, etc. This type of "modification" or "customization" of methods is referred to as <u>method combination.</u>

There are many different alternatives of method combination; the most prevalent types are daemons and around methods. Daemon methods are independent of the primary method and invoked either before or after it. Around methods are invoked and are given full responsibility as to when (if at all) to proceed with the remaining methods for the operation. If the around method does indeed call the remaining methods, it gains control again after their execution and may inspect and modify their results.

A full treatment of method combination alternatives is beyond the scope of this report, and may be found in reference **[Stallman 84]**.

3.5. Message Forwarding and Instance-Based Inheritance

The concept of inheritance discussed in the previous section, namely class-based inheritance, is the form of behavior inheritance most commonly found in modern object-oriented languages

Another type which does not usually receive much attention is termed instance-based inheritance or message forwarding[10]. Continuing with our aircraft example, we see that typically aircraft have onboard instrumentation. Those instruments are themselves objects that may have been defined as follows:

every INSTRUMENT is a LOCATION and
has an OWNER
every ALTIMETER is an INSTRUMENT and
has an ALTIMETER.ERROR

and so on for other instruments like airspeed indicators, vertical speed indicators, etc. These instruments belong to the class **LOCATION** but they also have an attribute called **OWNER** which we will assume points to the object to which they are attached. When a reading of indicated altitude is required from the altimeter, it must look at its true altitude, apply some white noise to it (say a normally distributed random variable with mean zero and standard deviation the value of **ALTIMETER.ERROR**) and return the result.

But, even though **ALTIMETER**, through its location **SUPERCLASS**, has its own **ATTRIBUTE**, it is not this value that we wish to use to compute indicated altitude, but rather that of its **OWNER**. Therefore, the **GET.ALTITUDE** method for **ALTIMETER** should return the altitude of its **OWNER**, whenever it has an **OWNER** capable of reporting its altitude, and its own altitude if it has no **OWNER**, or the **OWNER** object cannot report an altitude. In essence altimeters should forward the **GET.ALTITUDE** message to their **OWNER**, hence the term message forwarding.

It would be possible to achieve this by including **ALTIMETER** as a superclass of **AIRCRAFT.** Then **AIRCRAFT** would have inherited all the altimeter's behaviors and would also have shadowed the **GET.ALTITUDE** operation, as required. Even if the **ALTIMETER**'s method that computes the indicated altitude were to access the **ALTITUDE** attribute directly, (in violation of the minimum number of direct accesses principle), it would still work, since the **AIRCRAFT**'s **ALTITUDE** attribute would have shadowed the **ALTIMETER**'s identically-named slot.

Although correct from a strictly software point of view, (i.e., altimeters will indeed give the correct reading), such an implementation should be avoided at all costs. The main objection is that there is no real-world class relationship between altimeters and aircraft.

Secondly, altimeters are not only useful to aircraft, but may be useful to mountain climbers, for example. We would be forced to include **ALTIMETER** as a subclass for any object class that may require an altimeter reading. This is indeed what happens in SIMULA and similar languages, where the class hierarchy tree for a typical simulation object can be very deep. Forwarding the message to an object, which is the value of one of the forwarding object's attributes, is a powerful mechanism that is currently in Lisp-based simulations.

This forwarding operation can, of course, be manually coded in the appropriately-named method of the forwarding object. However, it would be simpler for the user if the forwarding operation were automatically handled by the compiler; additionally, the functional association between the two instances of objects is made more clear and visible. This could be achieved by a construct such as:

every ALTIMETER is an INSTRUMENT,
has an ALTIMETER ERROR,
and forwards GET.ALTITUDE to OWNER

It is interesting to note, by forwarding the **GET.ALTITUDE** message to the **OWNER** object, we have effectively overriden the **GET.ALTITUDE** methods of ALTIMETER's superclasses. However, since the value of the OWNER attribute is determined at run time, we have the important result that different methods may actually field the **GET.ALTITUDE** message for different instances of **ALTIMETER**, depending on who owns a particular inheritance alone.

Object-Oriented Distributed Simulation

A complete behavior inheritance scheme can be constructed by means of the message-forwarding feature, if a "standard" attribute is made to be present in every object which indicates the "parenthood" relation- ship. The ROSS language uses this mechanism as the behavior-inheritance mechanism, as opposed to the class-based mechanism of SIMULA, Smalltalk, and the Lisp Flavor System. Both approaches are workable and could be viewed as two different implementations of the same concept.

However, there are a number of important differences between them:

1. In the class-based inheritance concept, classes of objects (e.g., **AIRCRAFT**, **ALTIMETER**, etc.) are qualitatively different from instances of objects of that class. Even though they may be objects themselves (i.e., class **AIRCRAFT** may be implemented as an instance of class **CLASS.DESCRIPTION**), they do not support messages that the objects they describe support (e.g., the instance of **CLASS.DESCRIPTION** describing the **AIRCRAFT** class does not support the **ALTITUDE** method or any other method supported by aircraft instances).

   In the instance-based inheritance concept, classes are only quantitatively different from instances of objects. In ROSS for example, objects whose **OFFSPRING** attribute is a non-empty set are classes, otherwise, they are instances (the terminology used in ROSS is generic and specific objects).

2. Each concept implies a different view of the world. Class-based inheritance views the class behavior as being an integral, inalienable part of each object. Once an object has been instantiated, we can alter its behavior only by altering the contents of its attributes, but we could not change its class structure and makeup.

   In contrast, instance-based inheritance views classes as sets. Objects inherit the class behavior by belonging to the set. At any point, we may add an object to a class or remove it from another class; an aircraft can turn into a tank by a simple modification of one of its attributes.

3.6 Desirability of Object-oriented Programming

There are three reasons why Object-Oriented Programming should be considered as a technique for implementing system simulation:

1. The abstraction barriers provided by Object-Oriented programming enhance the desirable modularity in programming that makes large systems manageable.

2. The software architecture that results from programming in Object-Oriented style often match the experiential perception of the system being simulated; this simplifies the mapping between elements of the real system being simulated and the corresponding software element simulating them, and between the flow of causality in the simulated system and the flow of control in the simulation.

3. The Active-Object/Message-Passing paradigm of computation seems to lead to a practical and effective way of implementing concurrent, synchronized multiprocessing, and indeed has been proposed as an approach to concurrent simulations **[Jefferson 85]**.

On the other hand, it must be made very clear that there is nothing in an object-oriented program that could not be coded in a conventional way, much in the same way that there is no SIMSCRIPT program that could not be implemented in FORTRAN, BASIC, or, for that matter, machine code. Of course, any ditch dug with a steam shovel could also be dug with a teaspoon, though perhaps not in one lifetime.

The use of an object orientation holds obvious promise for the construction of large simulation models. For example, the combat models SCSS, SLAATS and CASTFOREM have already been built using traditional SIMSCRIPT II.5 to implement an object-message approach. This approach becomes even more obvious when dealing with communications simulations, such as NETWORK II.5 **[Garrison 1984]**.

However, the net advantage of an object-oriented framework in either a single or multi-processor environment will become obvious only when the proper tool is placed in the hands of experienced simulation modelers. A 50-line demonstration model in any language is not an accurate predictor of its utility in writing a 50,000-line program.

CHAPTER 4: A LANGUAGE FOR CONCURRENT SIMULATION

This chapter describes a proposed Language for Concurrent Simulation that could be used on MIMD systems. The LCS design encompasses many of the concepts of object-oriented programming described in the previous chapter. It also inherits many of the properties of the SIMSCRIPT II.5 language, notably including the direct support of SIMSCRIPT's higher-level simulation constructs,

In deciding how to mesh the concepts of object-oriented programming with existing simulation capabilities, we have to consider the following issues of potential compatability issues:

1. Maintaining upward compatibility with the existing simulation concepts and training.

2. Maintaining maximum commonality between the standard implementation on a single CPU and the parallel processing environment of an MIMD system, such as the Caltech HyperCube.

3. Keeping the distinguishing features and concepts of the language (processes, events, etc).

4. Maintaining the current syntactical flavor of the language.

The basic building block of LCS is called an object, which is similar to a SIMSCRIPT <u>temporary entity</u> with message receiving properties added. In their current implementation, temporary entities exhibit many of the fundamental traits of objects. In particular, they have an internal state description, (attributes) and operations can be defined on them.

At the same time, the concept of an object can be used as an infrastructure for defining many of the existing SIMSCRIPT artifacts. This would allow gradual re-implementation of the great majority of SIMSCRIPT internal constructs (like sets, I/O streams, etc), in the object-oriented style. Throughout the remainder of this chapter, the terms "object" and "entity" will be taken refer to LCS objects.

4.1 Background of SIMSCRIPT II.5

The original SIMSCRIPT programming language was developed at the Rand Coporation for the U.S. Air Force in 1962. It was one of the earliest general-purpose simulation languages, along with GPSS and CSL. In some ways, SIMSCRIPT I **[Markowitz 1963]** was also among the earliest languages to incorporate object-oriented concepts. For example, a SIMSCRIPT event is an active object with a limited number of behaviors.

The later SIMSCRIPT II **[Kiviat 1968]** progressed even further towards a true object-oriented language by providing a message-like approach for querying and changing object attributes through use of monitored variables. Each such attribute has both data and program associated with it, and two "methods" are automatically defined -- one for accessing the value, and one for changing the value. As with other object-oriented languages, no syntatic distinction is made between accessing an attribute or the value returned by a method routine.

This monitoring capability is also used for defining additional behaviors for statistics gathering. Such statistics include a user-specified list of properties and qualifiers.

Although the original SIMSCRIPT was a translator to FORTRAN, the current SIMSCRIPT II.5 is now a full compiler implemented on most mainframes and minicomputers. A significant enhancement is the process feature **[Russell 1983]**, somewhat based on GPSS activity blocks, which allows specification of sequential actions for a simulation object. This enhancement was originally developed a decade ago to support a major combat model, SCSS, that is still in use today.

SIMSCRIPT II.5 is specified by AMIP software development standards **[AMMO 1983]** as the standard Army simulation language. The language has been used for a large number of major military models **[CACI 1985]**. It is also used in modeling manufacturing, transportation, and communications problems. It has gained recent importance in the implementation of general-purpose computer systems simulators, such as NETWORK II.5 and ECSS.

SIMSCRIPT is unusual in that it is one of the few general-interest languages that is defined and distributed by a single company. This allows CACI to efficiently enhance, support and teach the language, without hindrance of outside committees or organizations. Several hundred simulation professionals each year attend CACI's course, "Simulation and Model Building Simplified with SIMSCRIPT II.5." The company also maintains an active university program to encourage the teaching of SIMSCRIPT II.5 at both the graduate and undergraduate level.

Late in 1984, CACI was hired by JPL to study the requirements for developing a parallel-processing simulation language. This report is a summary of the results of that study.

4.2 Objects in LCS

An entity in LCS entities has attributes and retains all the capabilities of temporary entities in SIMSCRIPT II.5. The syntax for defining them and instantiating them will also remain similar by allowing new types of clauses, but keeping all the existing ones as they are now. A high degree of compatibility with current code is thus maintained. The proposed syntax is:

```
every ENTITY.NAME
     [has ATTRIBUTE, [ ATTRIBUTE2, ...] ]
     [is  OBJECT, [ OBJECT2, ...] ]
     [refers to OBJECTA, [ OBJECTB, ... ] ]
```

Some examples of this are:

```
every LOCATION has a LAT, a LON, and an ALTITUDE, and
     define LAT, LON, ALTITUDE as real variables

every INSTRUMENT is a LOCATION and
      has an OWNER
define OWNER as an object variable
```

The use of the **refers to** clause will be discussed in Section 4.7.

Another subtlety in the interpretation of the **DEFINE** statement for entity attributes can be identified. Since **LAT, LON, ALT,** etc., are now local to the objects and can be accessed only through appropriate messages, we are implicitly specifying that a **LAT** message to an object of type **LOCATION** is a **REAL** function (i.e., it returns a floating point value). These definitions should be local to the entity we are currently defining. In the best case the user should be able to say:

```
every SHIP has a MAST, and a FUEL
     define FUEL  as a real variable
     define MAST as an integer variable
every CAR has a MAKE and a FUEL
     define FUEL as an integer variable
     define MAKE as  a text variable
```

That is not to say it is reasonable for the user to want to do this. It should be possible, however, for an attribute name to have distinct typing for each temporary entity class. Notice that there is a vast difference between the above and

```
every VEHICLE has a FUEL
     define FUEL as a real variable
every SHIP is a VEHICLE
          and has a MAST
     define MAST as an integer variable
every CAR is a VEHICLE
          and has a MAKE
     define MAKE as a text variable
```

The inevitable consequence is that when later in the code the message **FUEL** is sent to an object **X**, it is impossible for the compiler to tell whether the message will return an integer or a real value. It may be desirable to require static tying of **X** in such cases, or for the compiler to encourage the user to define a more consistent generic usage of the attribute **FUEL.**

Variable typing will also create problems with respect to attribute merging. In particular, when two classes have the same attribute name and each has declared it to be a different type, it is not clear what should happen when a third class is built on both of the first two. One approach would be to use the most general of the two declarations. This would mean that if one was declared integer and the other real, the composite class would implicitly define this attribute as real. From a practical standpoint, however, it would be more appropriate for the LCS environment to signal a compile-time error if this occurs.

4.2.1 Declaration of Objects

As noted in the previous section, the structure of a global LCS object is declared in the PREAMBLE, much as for a SIMSCRIPT temporary entity or a Pascal record. These attributes are part of each instance of the object and are accessible in any module of the program, class. Thus, the attributes must be declared external to all such modules.

However, there may be some state properties for the object which should not be globally defined and accessible, but instead available only in procedures associated with the particular object class. We will distinguish the two types of data values by referring to the former as public attributes, and the latter as private attributes.

As noted earlier, public attributes are declared in the section of the program that defines the specifications of the object. If the object is global in scope, the specifications are in the program preamble. However, the declaration of private attributes is associated with the IMPLEMENTATION of the object, and thus is declared in conjunction with the appropriate routines for implementing the object's behaviors.

Not suprisingly, the implementation of an object is defined with the **object** statement, which in some ways is similar to the **event** or **process** statement of SIMSCRIPT II.5. The section may be followed by one or more variable definition statements. Such variables will be declared as private attributes, and space will be reserved in each instance of the object to hold these attributes. However, only methods associated with the particular object class will be able to access these attributes. The section is termianted -- as is any routine or module -- by the **end** statement.

The implementation section may also include one or more executable statements, which will be executed after the instantiation of the object. This section will commonly be used to define to default values for instances of the object.

For example, consider a sample specification of an **AIRPLANE** object:

```
every AIRPLANE
     has X, Y, Z, SPEED, HEADING and FLIGHT.STATUS
```

This defines six public attributes for all objects of class **AIRPLANE.**

Next, consider, the sample implementation section

```
object AIRPLANE
     define CABIN.PRESSURE as real variable
     define FOOD.SERVICE.STATUS as text variable

     FLIGHT.STATUS = "Grounded"
     FOOD.SERVICE.STATUS = "Not served"

end
```

This declares two private attributes for the object, as well as default values for both public and private attributes.

After the necessary initialization code, the section may include the corresponding method routines for the object's behaviors. This allows grouping of the implementation of simple objects into one source file.

For more complex objects, methods may also be specified outside the object declaration using, in this case, the **for <object.type>** modifier. (See Section 4.3)

The object block may also include a declaration of other objects, as discussed in Section 4.8.

4.2.2 Referencing Object Attributes

In SIMSCRIPT and its successors, the attributes of an entity are referenced by listing the attribute name, followed by the entity pointer in parentheses, as in:

WEIGHT(PLANE)

This provides a comfortable analogy to ordinary arrays, which in fact is used to implement a slightly different type of entity (a permanent entity) which has identical syntax but is implemented as an array. This similarity also facilitates the monitoring of attributes, since the syntax is indistinguishable from calling procedure **WEIGHT** with argument **PLANE.**

Unfortunately, this syntatic overlap has two obvious disadvantages. First, there's no way for an object to directly have an array as one of its attributes - a rare but not unimportant requirement. Secondly, the monitored variable can receive the entity pointer as an argument, but no other arguments are possible.

If LCS were to be used by a significant number of SIMSCRIPT II.5 programmers and programs, it might be desirable to "grandfather" this existing syntax. However, a more flexible approach is needed to allow arguments (subscripts) for object routines (arrays).

LCS adopts the colon (":") as a separator between the object pointer from its attribute. Thus, the preceding example would become

PLANE:WEIGHT

for the attribute **WEIGHT** of an instance of **AIRPLANE** called **PLANE.**[11]

An array attribute of the object could then be expressed as

PLANE:RPM(ENGINE.NO)

and so forth.[12]

The colon delimiter of LCS has the same role as the dot delimiter of Simula, as in

PLANE.WEIGHT

This syntax, however, cannot be used in Pascal or C because in those languages it implies that **PLANE** is a static, rather than dynamic, data structure. Instead, Pascal (Object Pascal) would express the **PLANE:WEIGHT** of LCS by first de-refencing the pointer, as in

PLANE^.WEIGHT

In C (or C++), the arrow delimiter is used, as in

AIRPLANE->WEIGHT

Even in those languages that provide for both approaches, the vast majority of object references will be to dynamic data structures. Therefore, omitting a syntax for static object references proves only a minor reduction in flexibility for typical simulation programming. However, it provides a sizable benefit in conceptual clarity and ease of use.

As noted in both C++ and Object Pascal, the specification of the object is redundant within the corresponding method routine (see Section 4.3). Therefore, in a routine specifically for object **AIRPLANE**, the following references would be unambiguous and equivalent to previous examples:

WEIGHT
RPM(ENGINE.NO)

This is similar to the SIMSCRIPT II.5 construct of <u>implied subscripting</u>, but avoids all of the problems associated with its unrestricted use. It also allows public attributes, private attributes, and method variables (section 4.3) to be used interchangeably within a method -- a healthy freedom, given that the distinctions may change as the system is implemented.

4.2.3 Collections of Objects

It is very rare that a program will concern itself with only a single instance of an object. Instead, most object classes will be represented by multiple instances, which the user wishes to keep track of and manipulate, either as a group, or as one of several subgroups.

Smalltalk refers to such a group or subgroup as a collection of objects. The requirements for a collection may vary widely from application to application, or even within the same program. In some cases, the order of the objects is important; the order within the collection may be defined by the order of entry, or by some attribute of the object. Many collections -- such as a dictionary of code phrases -- are accessed primarily by looking up an object with an attribute matching a known key. Other collections have no order or structure whatsoever.

The primary strength of the SIMSCRIPT I language was in its view of the world terms of objects, properties and collections, or, in SIMSCRIPT terms, "entities, attributes and sets." A SIMSCRIPT set is a particular type of ordered collection. It may be implemented as either a singly- or doubly-linked list, with a first-in, first-out or last-in, last-out ordering. The user is also allowed to specify a series of ranking criteria for ordering of entities with the set (See **[Mullarney 1983]**).

An important requirement of a modern object-oriented language is the provision for a number of different types of collections, both of standard system forms and those defined by the user. When used in a parallel-processing environment, such collections should also address the issues of concurrent searching and non-deterministic ordering. This will be discussed in Chapter 5.

4.2.4 Object Variables

The most fundamental change brought about by introducing object oriented concepts is the need for dynamically-typed variables. In essence, we need a variable type that can hold an arbitrary object and a way to determine at run time what type of object is pointed to by the variable.

As was noted in Chapter 3, dynamic typing is not a pre-requisite of object-oriented programming. On the other hand, it is soon apparent that strong typing can be very restrictive. Unless dynamically typed variables exist, one would generally find it impossible to perform iterative calculations on a set of objects unless they were all of the same class.

For example, in a strongly-typed language, code such as

```
for each X in SET.OF.OBJECTS
do
     ask X LATITUDE yielding LAT
          .
          .
          .
loop
```

could not compile correctly unless we tell the compiler that **SET.OF.OBJECTS** can only contain objects of a certain type (say **AIRPLANE**). Then **X** could implicitly become a variable of type **AIRPLANE** at least for the duration of the loop. In that case, the latitude message for AIRPLANE could be hardwired into the code at compile time. This would sacrifice much of the flexibility that characterizes object-oriented programming.

For purposes of commonality, the statically typed variables that are now in existence can co-exist with dynamically typed ones. The syntax for dynamically typed variables will be

define X as an object variable

which is analogous to

define Y as an integer variable

This syntax provides a possibility which is open to a number of interpretations:

define X as a LOCATION variable

This statement can mean either:

1. **X** can point to any object that is of type **LOCATION** or has **LOCATION** as one of its superclasses. This is a form of dynamic typing, except that it is more restrictive. We will call it <u>deep typing</u>.

2. **X** can only point to an object of class LOCATION but not to one that has **LOCATION** as a superclass. This is a form of static typing for object variables. We'll call this shallow typing.

Whichever of the above interpretations we chose to implement, the statement can be as for two possible actions from the compiler.

1. It constitutes a promise to the compiler, thus giving it permission to optimize the code under the assumption that **X** will indeed always point to a legal object type, or

2. It is a request for run-time error signaling if any legal object type is assigned to **X**.

Unfortunately, we might want to select a different interpretation depending on where in the code the statement appears. For instance, if it appears in the preamble, it is likely that deep typing combined with the error-checking request might be the most appropriate interpretation. As an example, for the code:

every CAR has a DRIVER
define DRIVER as a PERSON variable

It is almost certainly best to create runtime checks so that the driver of a car is never assigned an object which does not have **PERSON** somewhere in its class structure.

4.2.5 Instantiation of Objects

An instance of an object is allocated in LCS by the **create** statement. Thus, the statement

create AIRPLANE

would allocate a new instance of class **AIRPLANE** and assign a pointer to that instance to the variable of the same name. The **called** qualifier can be used to assign the object pointer to a different variable, as in

create AIRPLANE called TWA747

The creation of temporary entities should also remain upwards compatible with SIMSCRIPT II.5. Three optional clauses will also be available.

```
CREATE A class.name CALLED name
     with attribute.1 value.1
       [,{attribute.2 value.2 ...}]
```

The **WITH** clause simply specifies that the following pairs are alternating attribute names and initial values for the new entity.

The subject of object deallocation is one of some controversy within the field of modern programming languages. One school of thought holds that the user should not be responsible for deallocating the memory associated with an instance. Instead, the system should "know" when the instance is no longer being used and return the memeory at that time. This approach is taken by Smalltalk-80 and implementation of Lisp.

This design point eliminates many common programming errors. For example, an object could be deallocated while it is still a member of a linked list, thus corrupting the entire collection. Or the object may be "known" to many objects, any one of which could deallocate the object without informing the other objects.

On the other hand, from a practical standpoint automatic deallocation poses a number of serious implementation problems. As outlined in **[Goldberg 1983]**, there are two traditional approaches to automatic deallocation. The first approach uses a count of the number of references to an object. When the count returns to zero, the object is deallocated. This fails to properly handle certain cyclic data structures, and also significantly slows the assignment of object pointers, since each such assignment means decrementing one counter and incrementing another.

The alternate approach is to perform a periodic garbage collection, by marking those objects which are still known by some path from the root object. The remainder are assumed to be "lost" and thus can be deallocated. This garbage collection can be very slow, particularly in a virtual memory environment. This also results in a large amount of wasted memory, which may be unacceptable on a smaller non-virtual machine.

The author beleives that manual deallocation offers a viable alternative, when the proper protections are provided. Such protection would include:

* Detection of accesses to deallocated objects
* Refusing to deallocate an object in a collection
* Infrequent re-use of memory to maximize access detection

Such an approach has been used for the past five years in one implementation of SIMSCRIPT II.5 ([**West 1984**]), by providing an explicit checkout mode that also detects other common data referencing problems. In actual use at hundreds of sites, for models ranging from 200 to 200,000 lines, this approach has been proven to be of pratical use without the performance disadvantages of automatic deallocation.

An object instance can be deallocated with the **destroy** statement, as in

```
destroy AIRPLANE
```

Each class of objects also has user- and system-defined **destroy** methods (see Section 4.3) to take care of cleanly terminating the object's existence. The user might wish to manually remove the object from any collections it is in, or print a trace of the action to a debugging file. The system-defined method would take care of deallocating any compounded data structures -- such as string or array attributes -- then would call the appropriate system memory manager routine.

4.3 Method Routines

A method routine specifies the action to be taken for a particular message of a given object class. The sending of a message to an object (Section 4.4) will eventually cause the program to execute one (or more) method routines corresponding to the message specifier and object type.

4.3.1 Declaration of Methods

The specification of a method for a class of temporary entities can be syntactically similar to the current function and routine definition.

```
method MESSAGE.NAME [ for CLASS.NAME ]
     [given ARGUMENT1 [, ARGUMENT2 ...] ]
     [yielding VALUE1 [, VALUE2 ...] ]

''   Method code

end
```

For example,

```
method DISTANCE for LOCATION given OTHER.ENTITY
     define OTHER.ENTITY as a LOCATION variable
     define DX, DY, OTHER.LON, OTHER.LAT as real variables

          ask OTHER.ENTITY LON OTHER.LON
          ask OTHER ENTITY LAT OTHER.LAT
          DX = (LON - OTHER.LON) * 60.0
                * COS.F (0.5 * (LAT + OTHER.LAT))
          DY = (LAT - OTHER.LAT) * 60.0
          return with SQRT.F(DX**2 + DY**2)
end
```

Note the addition of the **for ENTITYTYPE** keyword, indicating the object type. As noted in Section 4.2, if the method routine is within an **object** block, these keywords are optional. **Given** and **yielding** keywords can of course be specified for methods just as they can be specified for functional routines.

Here is a good place to point out that **DISTANCE** is not a global symbol as is the case with the current SIMSCRIPT II.5 functionality that can be given to temporary entities. The symbol can overloaded and reused (say for **LOCATION** objects) and no conflict occurs. Each object has its appropriate method for **DISTANCE** and a combination of static and dynamic checking will assure that it is used. Good programming technique would suggest that the same name be re-used only for similar functions and parameters, but this is by no means a language requirement.

As is evident in the example, within an entity's method, its attributes can be accessed DIRECTLY just like variable references. This is similar to the way that a SIMSCRIPT II.5 process can access its attributes within its process routine.

As a convention, the variable **SELF.V** will be defined within any method and will always point to the instance of the entity on behalf of which the method has been invoked. This allows methods to send messages to the same object. For example, in the **DISTANCE** method illustrated above, we could have used

```
ask SELF.V CURRENT.COURSE yielding ROUTE.LIST
```

to obtain information derived from the object's state.

4.3.2 Arguments to Methods

Unlike Fortran, Pascal, C and other languages, all arguments to LCS routines are passed by value. This means that the actual of the variable is communicated in the argument frame.

Arguments may include any one of the standard scalar types:

* integer
* floating point
* character
* string
* enumerated type

where the last type also encompasses a variety of derived types, such as messages and Boolean values.

For support of object-oriented programming, acceptable argument types must also include:

* An object pointer

However, the acceptable arguments to a method routine do not include:

* An object instance
* The address of a scalar quantity

as are allowed and, in fact, encouraged by modern structured languages and their object-oriented derivatives, such as Simula, Object Pascal and C++.

The "address of" construct is frequently used as a crutch for allowing a routine to return a number of values. However, it has extremely undesirable consequences when associated with a no-shared-memory MIMD parallel processing system, and is even worse when operation is under the Time Warp operating system is considered (see Chapter 5).

Instead, LCS allows any number of arguments to be returned by value. This construct is, in the author's experience, unique to the family of languages derived from SIMSCRIPT I. As with the arguments **given** to a method routine, the value of the arguments returned to the caller through the **yielding** construct are included in the argument frame and unstacked by the calling object.

4.3.3 Local Variables In Object Methods

As with any routine, a method may define its own variables. These variables are declared within the method routine with the **define** statement.

LCS is a member of the modern block-structured family of languages, and thus is fully recursive, so local variables are normally recursive in nature. Subsequent entries to the method will find the variables initialized to null values. The variables associated with a particular invocation of a method are not accessible outside that invocation, and are not retained after the completion of the invocation.

However, different methods operating on the same instance of an object may communicate themselves through either the public or private attributes of that instance. The private attributes are, in fact, intended to allow the implementation of an object's methods to internally share information without affecting the externally known specifications of the object.

Applications may be required where a method may need to share information between differing instances of the same class of object. For example, a queuing method could increment a local counter to provide a unique identifier for each instance of an object in the queue. The counter could be declared using the **static** keyword as used in C, which is similar to the **saved** keyword of SIMSCRIPT II.5. Such a variable, as in:

define COUNTER as integer static

would retain its value between subsequent entries to the method routine.

Both the arguments and local variables of a method routine must be distinct from the attributes and messages defined for the corresponding object (or its superclasses). This requirement, suggested by **[Tesler 1985b]** for Object Pascal, avoids a common ambiguity and avoids the awkward work-around required by the implementation of SIMSCRIPT II.5 events and processes.

Local variables are truly temporary, existing on the stack and popped upon exit from the method; they may be saved, however, while the method is in progress, as with a method that requires an interval of simulated time (Section 4.6.1). This treatment emphasizes that methods should be autonomous pieces of code without any memory of previous invocations and avoids programming errors due to inadvertent side-affecting among methods.

4.4 Message Passing

Three mechanisms exist for accessing the functionality of an LCS object. They are:

* Implicit (colon-delimited)
* The **ask** statement
* The **tell** statement

The implicit references are appropriate for messages that accept or return a single function value for use as part of an expression. The latter two statements can be used with arbitrary calling sequences, and are essentially similar, except as noted in Section 4.4.1.

The syntax of the **ASK** statement is:

```
ask OBJECT.VARIABLE [to] message
     [given {ARGUMENT.1...}]
     [yielding {VALUE.1,...}]
```

For example, if **BOS** and **PVD** are of class **LOCATION** and **DIST** is a real number,

```
ask BOS DISTANCE given PVD yielding DIST
```

would be the way to compute the distance between Boston and Providence (assuming the flat earth model implicit in the definition of **DISTANCE** above).

Because the LCS language is intended for simulation, messages often will be queued for execution at a future time. For example, when simulating a communications network, the arrival of a piece of information will occur at a finite interval in the future. To facilitate the transfer of such messages, the **ask** (and **tell**) statements may include an optional time of execution, as in

```
ASK object [TO] message.name AT sim.time
```

Language purists might object to this dilution of the generality of the object-message paradigm. But the alternative for this case would be to invent a fictitious event just to send this message, or to include the start time as an argument to a process-type object, which would **wait** the remaining time.

The flexibility this adds strongly suggests that this should be included. More importantly, many existing distributed approaches require a language-level time stamp to coordinate asynchronous time. "Hiding" the event time in the message arguments would only frustrate such coordination efforts and increase the chance of deadlock or rollbacks.

4.4.1 Message Synchronization

In a single-processor environment, most object-oriented systems implement message-passing analogously to a function call. Sending a message to an object (e.g. **TELL TANK ENGAGE**) means that the program or support system will find the appropriate method routine for that message/object combination and transfer control to that routine. When the method is complete, control will be returned to the routine that originally sent the message, perhaps with one or more return variables (**ASK PLANE POSITION YIELDING X,Y**).

When the same program is running in a distributed environment, this assumption of sequential action is unduly restrictive. Sending a message to another processor requires a certain amount of time, and a faithful reproduction of the single-CPU case would require that the requesting object wait until the method completes and the acknowledgement message is received. Perhaps the CPU can be put to other uses while awaiting the return message.

Under the Time Warp operating system, this restriction becomes much worse. The sending object may be at a different virtual time than the receiving object. If the sending object is further ahead in time, it must wait until the receiving object catches up, and THEN wait for the method to complete execution.

This suggests that there may be a requirement for two types of messages in a LCS. A synchronous message is one that behaves identically in a single- or multi-CPU system; the sender does not continue execution until the receiver's corresponding method is complete. If we think of messages as a form of electronic mail, then this is analogous to sending a "registered letter." The completion of the method automatically forwards a "return receipt" to the sender, which is merely a special form of message acknowledging the original message.

Some messages will always be synchronous. For example, any message asking for **YIELDING** values must be synchronous, because future computations are likely to use those value. Similarly, any message that returns a function value is synchronous.[13]

In other cases, however, the sending routine doesn't care what the recipient does with the message. If the gamesmaster of the multi-player simulation is sending out a new set of engagement rules, there may be no reason to wait for the message to be processed. In fact, if the message is going to 400 units, it would be impractical to send one message, wait for it to be processed, and so proceed sequentially for all 400 units.

In a virtual-time environment, the situation is further complicated by the prospect of waiting for laggard units to catch up to the gamesmaster's virtual time. The mere act of updating engagement rules could freeze the gamesmaster's object for several (elapsed) hours.

Instead, an asynchronous message is needed -- thus allowing the sending object to continue without further waiting. In a single-CPU system the distinction between synchronous and asynchronous messages will be unimportant. With a distributed simulation, extensive use generally of asynchronous messages will allow maximum possible utilization of the multiple CPU's.

Distributed computation and communications systems -- whether electronic mail or the Time Warp operating system -- tend to emphasize asynchronous message-passing for obvious reasons of efficiency. The raises the question: Is automatic message synchronization really necessary? After all, it can always implemented by waiting for an asynchronous response message.

However, certain concepts, particularly in a resource-oriented simulation, are more naturally implemented through a series of synchronous messages. For example,

```
ask LOGISTICS AIRLIFT yielding TRANSPORT
tell TRANSPORT GOTO(PICKUP.POINT)
```

The second message cannot be executed until the first has been executed. Even if both actions take place at the same simulated time, the sequential order within that simulated time must be maintained.

Because the most significant side-effects of (non)synchronization will occur in the sending object, it seems natural to make the distinction when sending the message. In LCS, the keyword **ASK** is used to send synchronous message, and **TELL** is used for an asynchronous one. Attempting to **TELL** for a **YIELDING** variable will result in a compile-time error.

4.4.2 Message Side-effects

The implementors of the Time Warp operating system have proposed a contrasting dichotomy of message types, relating to message side-effects **[Beckman 1984]**. In a single-CPU system the distinction is meaningless, as is the case with message synchronization. For that matter, if a message is received by an object to be executed in some future time, the two types are equivalent.

However, the distinction has been postulated for efficient implementation of the Time Warp rollback mechanism. In particular, the set of possible messages has been divided so that receipt of certain messages will not necessarily cause a time fault and rollback.

A query message is defined as one that does not change the state of the object. As its name suggests, the most common application will be for messages that inquire as to the state of the receiving object. A query message received from an object's past will be satisfied by looking up a saved state of the object. By definition, a query message cannot cause a rollback.

The most interesting (if not most frequent) class of messages will be those that do change the object's state, which are Time Warp refers to as event messages. Any message that MIGHT change the object's state -- such as the **GOTO** message -- must be declared as an event message, so that the receipt of an event message for some previous time will cause a Time Warp rollback. As currently implemented, this occurs even when the state is unchanged by the method. For example, a method that checks for something to do might find nothing, and thus leave the state unchanged -- but it would still cause a rollback.

However, a query method that changes the state of the object implies a logic error, or at least requires an immediate rollback. A query method also cannot send itself on an event message and would jeopardize time causality if allowed to send other objects event messages.

Unlike message synchronization, the distinction as to message side-effects clearly belongs with the recipient method, not the sending object. In most cases, the distinction can be discerned by the compiler and/or programming environment: if the method modifies attributes directly, or sends itself event messages, then it is an event method. Otherwise, it can be classified as the less-dangerous query method. Associating the distinction with the method definition (instead of the message dispatching) is also consistent with the object-oriented principle of hiding the implementation of the message behavior from those who use it.

The distinction between query and event messages can be seen as a difference between "read only" and "write only" state operations. However, this restriction of the Time Warp operating system seems too stringent to impose as a general language requirement. In addition, methods that may at first appear to be queries -- such as is found in SIMSCRIPT's monitored variables -- may actually end up having intended side-effects, such as the tabulation of message statistics.

Therefore, it seems undesirable to force the LCS user to manually declare each method as one type or the other. Manual intervention may be desirable, however, in unwinding cyclic message dependencies, perhaps through use of an interactive linker. Alternatively, the environment can assume that any method that sends a message is an event method.

4.5 Class-based Inheritance Of Object Behaviors

The implementation of behavior inheritance in LCS should be class-based, since it is clearly a more flexible and powerful alternative to instance-based inheritance. In addition however, the user should be given the facility to establish instance associations, as is noted later in this chapter.

A class-based behavior inheritance is declared with the **is** clause in the definition of an entity. As an example an AIRPLANE object may be defined as follows:

```
every LOCATION has a LAT, LON, ALTITUDE
every MOVING.OBJECT is a LOCATION and
     has NORTH.SPEED, EAST.SPEED and VERTICAL.SPEED
every FLYING.OBJECT is a MOVING.OBJECT and
     has a BANK.ANGLE and a LONGITUDINAL.ACCELERATION
every AIRPLANE is a FLYING.OBJECT and
     has MASS, POWER.LEVEL, LIFT.TO.DRAG.COEFF
```

When an AIRPLANE is created it will have all the attributes of **LOCATION**, **MOVING.OBJECT**, and **FLYING.OBJECT** as well as those of **AIRPLANE**. The following code would therefore be legal[14]

```
create an AIRPLANE called TW611
     with ALTITUDE 10000.0,
     LON - 71.0
     LAT - 42.0
```

If a class includes two or more superclasses that have the same attribute, the two attributers are merged into the one (i.e., there is only one physical location where that attribute is stored for each instantiation of the top level class). Note that an object can include a superclass and also it can have attributes that are of that superclass type. It is important to differentiate between the two. For example,

every AIRPLANE is a LOCATION and a FLYING.OBJECT
and has an ORIGIN and a DESTINATION
define ORIGIN,DESTINATION as LOCATION variables

For whatever reasons, the user included both **FLYING.OBJECT** and **LOCATION** as superclasses of **AIRPLANE.** Still, an instance of **AIRPLANE** will only have a single slot for each of the attributes **LAT**, **LON**, and **ALTITUDE**, as if the **LOCATION** class was not specified in the definition. However, the **LOCATION** methods will take precedence over any corresponding **FLYING.OBJECT** methods, because of the order of declaration.

On the other hand, the **AIRPLANE**'s attributes **ORIGIN** and **DESTINATION** will continually be distinct objects and each will keep its own individual slot for each of its **LOCATION** attributes.

When determining the default behaviors for an object, the behaviors are prioritized in the order of declaration. In this example, any method not implemented for **AIRPLANE** will first be inherited from **LOCATION** and then **FLYING.OBJECT.** This hierarchy includes all the inherited behaviors of the superclass -- so that behaviors inherited by **LOCATION** will outrank any defined by **FLYING.OBJECT.**

This provides conceptual simplicity, but does not handle certain complex relationships. For example, consider

every FLOWN.OBJECT is a PILOT and an AIRPLANE

This would allow commands to the **FLOWN.OBJECT** to generally be routed to the corresponding **PILOT** class. However, for the statement

ask FLOWN.OBJECT YOUR.WEIGHT yielding LANDING.WEIGHT

The appropriate response would be to write a short method which explicitly references the corresponding method for the **AIRPLANE** superclass. This could be done by lexically typing the object pointer when asking the method, as in:

```
method YOUR.WEIGHT for FLOWN.OBJECT yielding MASS
     ask (AIRPLANE)SELF.V YOUR.WEIGHT yielding MASS
end
```

The type-casting syntax shown is similar to that for the C language.

4.6 Simulation Object Classes

Since event notices and processes are temporary entities they will automatically turn into message receiving objects in LCS. In fact, one can implement event notices and processes as:

```
every EVENT.E has a TIME.A,E UNIT.A and belongs to
     an EV.S
every PROCESS.E has IPC.A,STA.A,RSA.A, owns
     a RS.S and is an EVENT.E
```

Finally, processes and events can maintain their process and event routine respectively. This should be totally distinct from their methods since its invocation procedure will be completely different. Namely, these routines will automatically be invoked by the system timer as is done in SIMSCRIPT II.5, and process routines they should maintain their recursive save area as is currently the case.

4.6.1 Time-elapsing Methods

For objects that are subclasses of **PROCESS.E** (hereafter referred to as a "process"), a number of behaviors are automatically defined for the implementation of the process construct.

A method of process does not have to complete within a particular simulated time. Instead, one or more statements may cause simulated time to advance until a particular condition or conditions are met. Such a method is referred to as a **time-elapsing method**.

One obvious case of such time elapsing occurs with the **wait** statement,

```
wait 10 hours
```

Other time-elapsing statements may involve a request that contains an implicit wait, such as

```
request 1 unit TELLER
```

For conceptual and implementation reasons, time-elapsing statement may be used only within a method for a process. They may not be included within a method for an ordinary object, or in a non-object routine or function.

Although not specified in the definition of the SIMSCRIPT II.5 language **[Russell 1983]**, recent implementations have allowed inclusion of such statements in ordinary routines. Disallowing this might appear to be a serious restriction. However, most of the usages of such statement could be more cleanly handled by a common class-based behavior inheritance. One common example would be the following method (in which the trailing dots are used to indicate the implied pointer **MOVING.OBJECT:**):

```
method GOTO(LOCATION) for MOVING.OBJECT
     DX = LOCATION:X - X..
     DY = LOCATION:Y - Y..
     NSTEPS = SQRT.F(DX**2 + DY**2)/MAX.SPEED..
               /TIME STEP
     OLDX = X..
     OLDY = Y..
     for J = 1 to NSTEPS
     do
       FRAC = J/NSTEPS
       X.. = OLDX = FRAC*DX
       Y.. = OLDY = FRAC*DY
       wait TIME.STEP
     loop
end
```

An attempt to use a time-elapsing method on a non-process will have the same result as any other message/object mismatch: a compile, link, or run-time error will be produced, depending on where it is detected.

4.6.2 Object Synchronization

In Section 4.5.1, a distinction was made between messages that are executed synchronously on a parallel processing system. The same distinction is significant when considering methods involving changes in simulated time.

Consider the **GOTO** method described in the previous section. In some cases, the object may wish to initiate a **GOTO** without waiting for it to complete. This would surely be the case of a dispatcher for a fleet of planes, as follows:

```
for each ROUTE in SCHEDULE,
do
     tell ROUTE:AIRPLANE GOTO(ROUTE:DESTINATION)
loop
```

For a typical hub-and-spoke operation (e.g., Federal Express), a large number of planes need to be dispatched simultaneously, so the dispatcher doesn't care how long it takes for the corresponding method to execute.

However, the same method might be used by an **AIRPLANE** object to implement a more complex routing that involves a trip along a specified list of **LOCATION**s. A method **FLY.ROUTE** to implement this might look as follows:

```
method FLY.ROUTE (ROUTING) for AIRPLANE
     for each ROUTE/LOCATION in ROUTING
     do
          ask AIRPLANE GOTO(ROUTE.LOCATION)
     loop
end
```

Because the **AIRPLANE** object must wait for the **GOTO** to complete before starting the next one, the **ask** statement is used instead of the **tell** statement to assure synchronization. Note that the method doesn't distinguish whether it was called synchronously or asynchronously.

To summarize, when a message is sent to a time-elapsing method:

ask causes the sending object to **wait** until method completion
tell allows the sending object to continue immediately

It is easy to see that statements such as **request** and even **wait** can be implemented as an **ask** to a system-defined method for class **PROCESS.E**. This does, however, raise the unresolved issue of the behavior of non-**PROCESS** objects that perform a **request**.

4.7 Instance-based Behavior Inheritance

The **refers to** clause of the **every** statement defines an instance-based inheritance for objects of a given class. The clause specifies one or more classes of object for which certain behaviors and attributes are to be deferred.

In addition to having its own declared attributes and those obtained from class-based inheritance declared in an **is** clause, an object also inherits all those attributes of classes declared in the **refers to** clause. However, these attributes are not incorporated within the template for the newly-defined class. Instead, the template includes a pointer to a particular instance of the specified class.

For example, take the following declaration,

```
every VEHICLE
     has an ORIGIN and a DEST,
     refers to a VEHICLE.TYPE
every VEHICLE.TYPE
     has a FUEL.CAPACITY and a MAXIMUM.SPEED
```

When created, a **VEHICLE** should specify a **VEHICLE.TYPE**, as in the following

```
create a VEHICLE.TYPE called VT.M1 with MAXIMUM.SPEED = 60

create a VEHICLE called NEWTANK with VEHICLE.TYPE = VT.M1
```

Then a method for calculating trip length could include the following statements

```
method HOW.LONG for VEHICLE
     return with DISTANCE(ORIGIN, DEST) / MAX.SPEED
end
```

To minimize the chance of errors, one restriction must be placed on the use of attributes obtained through an instance-based inheritance. If the subclass of object inherits the methods associated with changing the values of the superobject, this inheritance would be a recipe for disaster.

For example, the statement

let TANK:MAXIMUM.SPEED = 70

would change the value of **MAXIMUM.SPEED** for all tanks of the corresponding **VEHICLE.TYPE** -- which, if the user thought about it, is probably not what (s)he wanted. As a consequence, objects do NOT inherit the methods associated with changing the public attributes of their instance-based superobjects.

Such an assignment can be explicitly made, of course, by a statement such as

let TANK:VEHICLE.TYPE:MAXIMUM.SPEED = 70

Of course, attempts to access any attribute or method associated with an instance-based inheritance will produce a run-time error if the reference object pointer has not been initialized.

4.7.1 Class Variables

As an example of how instance-based inheritance could be used, let us consider the standard properties associated with each class of objects. The standard properties -- both attributes and behaviors -- for each object class could be represented by an instance of a standard object template, or **TEMPLATE.E.**

Each instance of an object would then **refer to a TEMPLATE.E**, and will be created with the following psuedo-syntax.

```
create an AIRPLANE
     with TEMPLATE.E AIRPLANE.T
```

where **AIRPLANE.T** is a system-defined and initialized variable pointing to an instance of a **TEMPLATE.E.** This association is transparently included for the user and is not a part of his/her declaration of the **create** statement.

Through this mechanism, all instances of a particular object class would then automatically standard properties. For example, a number of standard attributes could be associated with a **TEMPLATE.E**, and thus, all instances of any class. These would include:

CLASS.A -- an integer indicating the class of an object (equal to the predefined constant **C.OBJECT**)
CLASS.NAME.A -- a string with the name of the class
SIZE.OF.A -- an integer specifying the size of an object instance
NUMBER.OF.A -- the number of current instances of the object
SUPERCLASS.A(class) -- a boolean value (in array) indicating whether the specified object type is a superclass of the object
ACCEPTS.A(message) -- a boolean value (in array) indicating whether the specified message type is defined for the object

The **TEMPLATE.E** is similar to the class "Metaclass" of Smalltalk-80. Acknowledging this debt, we can adopt the Smalltalk term class variable to refer to the attributes of the template.

4.8 Instance-oriented Modular Programming

Previous structured languages have established data hiding relationships for recursive (stack-based) data values based on a nested block structure.

For example, Algol-60 allows for a series of procedures to be nested within each other. Recursive variables for the "outer" procedure are known within inner procedures, as in the following example, which prints "1" and "2" to the **PRINTER** file:

```
BEGIN
FILE PRINTER (KIND=PRINTER);

PROCEDURE PRINT(A,B)
     REAL A,B
     BEGIN
     REAL X;
     X := A;
          BEGIN
          REAL Y;
          Y := B
          WRITE (PRINTER, /, X, Y);
          END INNER BLOCK;
     END OUTER BLOCK;

PRINT (1.0,2.0);
END
```

In the example, the variable **X** is lexically scoped to include both the inner and outer block; the variable is "known" in either block and be read or modified in either. However, the variable **Y** has a lexical scope that includes only the inner block. Both variables are not known outside the procedure **PRINT.**

Also, if the procedure **PRINT** were called from another point in the program, the values of **X** and **Y** would be completely distinct from the call listed above. This is because each entry to the procedure causes a new set of memory locations to be set aside and initialized for the variables. We can think of this as a form of stack-based data hiding for different activations of the procedure. (A variable declared **OWN** in Algol is common across all entries to the block, similar to the LCS concept of **STATIC** variables).

Simula, of course, is a dialect of Algol, and shares its fundamental block structure characteristics. Pascal is one of its descendants.

As noted earlier, later derivatives of Pascal have attempted to develop a more general construct. We will adopt the Modula-2 term "module" to refer to a group of related routines that access a set of values.

As with the outer Algol block, the module variables are lexically scoped -- the identifier name is known only within the routines of the module. However, any entry to one of these routines will modify the module variables. There is no data hiding between entries; the variables are lexically local to the module, but static between entries in the module.

LCS implements as instance-based data hiding, and object-oriented analog to the stack-based data hiding of Algol-60. It does so by allowing a definition of a hierarchy of objects (for program and data) comparable to the hierarchy of blocks (for program only) defined by Algol.

4.8.1 Declaration of child objects

Previously, we have dealt only with objects that are declared in the program preamble and are thus global in scope. These objects, can, naturally, be referred to as global objects.

However, new object classes can also be declared within an **object** routine, as follows

```
object AIRPLANE
     every AUTOPILOT has a COURSE, SPEED
```

In this case, a block structure of lexical scope is established, analogously to that in Algol-60. In LCS, the outer object (e.g. **AIRPLANE**) is called the parent object, while the inner object (**AUTOPILOT**) is called the child. Children, may, of course, beget children; there is no restriction that a child object must be declared within a global object definition. (It is clear that global objects are, in turn, merely children of the preamble.)

The objects, attributes, and messages for a child object are known only to its parent. Child objects may communicate with siblings -- other instances of the same class. They may also communicate with cousins -- instances of objects of a different class that share a common parent.[15]

Child objects also have access to the data and program of the parent, and can interrogate and change the parent's state. If limited to lexical scoping, this data hiding would be comparable to the module concept, in which the parent object represents a module (or unit, or package).

However, while the child objects have access to both the public and private attributes of the parent object, they so do only for the particular INSTANCE of the parent that created them. This is an instance-oriented data hiding for objects that is analgous to the activation-oriented data hiding of Algol procedures.

The following example may clarify this.

```
every PARENT has a VALUE
object PARENT
   entities include CHILD

   method DISPLAY given X
       VALUE = X
       create CHILD
       ask CHILD PRINT
   end
end
object CHILD
   method PRINT
       list VALUE
   end
end
main
       create a PARENT called PARENT_1
       create a PARENT called PARENT_2
       tell PARENT_1 DISPLAY (1)
       tell PARENT_2 DISPLAY (2)
end
```

would cause the output

```
VALUE = 1
VALUE = 2
```

to appear on the output device[16]

The number contained in variable **VALUE** for the instance **PARENT_1** is completely inaccessible to the **CHILD** of instance **PARENT_2**, and vice versa.

In fact, any parent object is also, necessarily, the superobject of an instance-based inheritance. Although not explicitly programmed as such by the user, the above code can be conceptualized as

```
every CHILD refers to a PARENT
     ...
create a CHILD with PARENT PARENT
     ...
method PRINT for CHILD
     list CHILD:PARENT:VALUE
end
```

If necessary, a number of instances of differing child objects can communicate through the attribute of their common parent object (instance). The instance of the parent objects and all the instances of its children are then referred to as an environment, as first proposed by **[Elias 1985]**. Unlike the earlier proposal, however, any number of instances of the same class of environment may exist in the simulation, and the creation of different environments is entirely data-driven.

Externally, the environment is visible to other instances and environments only through the external specifications (public attributes and methods) of the parent object. Environments, of course, can be nested withing environments. All objects in the simulation are, in fact, part of the global environment.

The availability of object environments offers a number of significant advantages in construction large simulation models for exeuction on MIMD systems:

1) Lexical data hiding within an environment separates the global specification of a parent object from its local implementation, as is true with structured language modules.

2) Instance-based data hiding restricts the use of side-effect programming and provides a structured framework for object-oriented programming.
3) A protocol is provided for specifying complex time-elapsing behaviors for the parent object, as noted in the next section.
4) An important clue is provided for distributing data across parallel CPU's. If objects were unscoped, any object could interact with any other object, offering almost no hope for discerning tightly-coupled objects. Even a module-like lexical scoping would mean that any INSTANCE of the same class of inner object could interact with all the common data of the outer object.

The use of instance-based environments for distributing data will be discussed further in Chapter 5.

4.8.2 Use of Child Objects

As noted previously, one use of the parent/child hierarchy is in implementing a complex behavior for the parent object. This is particularly true when the parent object is a process; conceptual and implementation considerations strongly suggest that an object executing a time-elapsing method accept only a restricted class of messages.

For example, the Smalltalk class **DelayedEvent** defines a limited protocol of messages that can be accepted:

* Set the condition for event sequencing
* Begin event when condition is reached
* Pause (interrupt) the event
* Resume an interrupted event
* Decide if sequenced before another **DelayedEvent**

Similarly, if a process is executing a time-elapsing method, it is reasonable to allow the process to respond without limit to query messages, as defined earlier. However, receipt of any event method other than **INTERRUPT** should result in a run-time error. Once interrupted, the method could only be **CANCELLED** or **RESUMED**.

Failure to impose this restriction would open a Pandora's Box of indeterminate states that cannot be shut. The restriction would not hamper the modeling of certain classes of "dumb" processes, such as a group of docile (non-balking) customers in a fast-food restaurant.

This would not be an acceptable restriction in more complex problems, such as modeling a trans-continental airplane flight through a network of air traffic control centers. Even if the plane sets its course as "due west" and then waits for six hours, it must be able to respond to a series of arbitrary commands during that flight -- as in a changed destination resulting from bad weather at the destination airport, or an order from headquarters to divert to pick up other passengers.

As suggested by the earlier example, this case could be supported by defining an **AUTOPILOT** as a child object within the **AIRPLANE** environment. A psuedo-code version of this would be:

```
object AIRPLANE
     define AUTO.SET as a boolean variable
     create an AUTOPILOT
          ...
     method FLY.TO(ROUTE)
          if AUTO.SET
               interrupt AUTOPILOT
               cancel AUTOPILOT
          endif
          tell AUTOPILOT FLY.TO(DESTINATION)
     end
end

method FLY.TO(ROUTE) for AUTOPILOT
     AUTO.SET = .TRUE
     for each LEG in ROUTE,
     do
          AIRPLANE:HEADING = LEG:COURSE
          wait LEG:TIME
          AIRPLANE:LOCATION = LEG:TERMINUS
     loop
     AUTO.SET = .FALSE
     tell AIRPLANE ARRIVAL
end
```

In this way, the **AIRPLANE** object is free to respond to messages from objects external to the environment. However, the **AUTOPILOT** object will continue along the specified route, changing the plane's coruse and location as it goes. Note that the **AUTOPILOT** explicitly modifies public attributes of its parent **AIRPLANE**, thus allowing it to arrive at its destination without any explicit intervention on its part.[17]

If the **AIRPLANE** receives a conflicting set of instructions, it can stop the **AUTOPILOT** from flying one course and start it flying another. When the route is completed, a message is sent to the **AIRPLANE** to allow it to perform the actions associated with arrival, such as landing at the airport.

An alternate mechansim may be desirable for the coordination of parent and child process interactions. Many classes of child processes may not need more than one method and its sole function can be implemented as part of its initialization code. Upon termination of that code, the process would be destroyed, but before destroying itself, the child should inform the parent of its completion, as in

```
ask AIRPLANE CHILD.DONE(AUTOPILOT)
destroy AUTOPILOT
```

A similar mechanism is used by the UNIX operating system to coordinate between a task and its subsidiary tasks. (These are referred to as the "parent process" and "child processes", consistent with LCS terminology.) As provided for in Version 7 and the Berkeley variants of UNIX, the completion and termination of a child process results in the C-langauge function call:

```
kill (parentid, SIGCLD);
```

where **kill** is the misnamed system routine to send a signal (message) to the task **parentid**, and **SIGCLD** is the standard signal for the "death" of a child. The parent process, in turn, is expected to "catch" the signal by the following system call, which suspends the parent until one of its children terminates:

```
childid = wait(&status)
```

In this case, **childid** is the process identifier of the terminated process, and **status** is used to pass a 16-bit status value returned by the child process.

The object-oriented analog would be

```
method CHILD.DONE given CHILD
    ...
end
```

Because the parent would receive the message before the child is destroyed, it can access all of its public attributes to determine its state. Unlike the UNIX example, however, the parent process does not have to explicitly wait for **CHILD.DONE**, but is free to continue with its business, receiving any message from inside or outside the environment.

4.9 Durable Objects

By modern standards, early computer systems had tiny amounts of memory. A standard IBM 360 configuration of the late 1960's had less main memory than a typical personal computer model sold today, such as a fully configured IBM XT or Apple Macintosh.

As a consequence, a distinction was made between the files stored on peripheral devices and a program's memory-resident data structures. Early systems stored files sequentially on tapes, but these were soon supplanted by random-access disk drives.

With the advent of gigabyte-level virtual memory systems and new techniques for taking advantage of such memory, it has been suggested that this dichotomy between files and data structures has by now become obsolete. This may be particularly true for a complex simulation, which has a number of requirements for data management.

Within a large simulation run, it may prove necessary to stop the program, analyze the current data structures and possibly rearrange their relationships. Alternate scenarios may be desired by capturing one set of relationships and re-playing the simulation from that point with different parameters.

From a more static viewpoint, the same global structure may exist across a number of simulation runs. The topology of a communications network, the terrain of a region, or the force structure of a military unit are often shared across a series of exeuctions of the same simulation -- or even different simulation models. The current approach is to store the relationships in human-readable format in a sequential data file, then rebuild the data structures each time the program is run.

A number of solutions have been developed to the problem of data management. Some systems support development of a data base through a series of standard applications programs, which then allow access to that data base via subroutine calls in the user's program.

However, to make the integration of files and data work, the user should only use one protocol for manipulating either type of data within his/her program. If a hashed collection allows location of an object with language statements such as

```
for each SYMBOL.ENTRY,
    with KEYWORD = USER
```

why should (s)he be forced to call a subroutine to look up a record in a data base, then manually transfer the data to a memory-resident data structure? The only practical distinction between an object instance and a database record is the durability of the data: the former exists until the program exits, while the latter remains until erased by an errant user or a hardware failure.

To address this problem, the concept of durable objects (or "durable entities") has been proposed by **[Mullarney 1982]**. The term was coined to distinguish the new type of object from existing data concepts of "temporary" entities (frequently created and destroyed) and "permanent" entities (usually created only once per run).

As proposed by Mullarney, the instances of a class of durable objects would be stored on a disk file and then loaded into memory when used within a program. As with other virtual and swapped memory algorithms, changes in the memory-resident objects would be translated by the hardware into changes in the disk-resident data structures, either incrementally or at the completion of the run.

The original implementation proposal envisioned an MS-DOS file stored on the hard disk of an IBM XT. The format of the file would be compatible with the segment architecture of the XT's 8088 chip, which uses the upper 16 bits of an address to specify a segment number and the lower 16 bits for a segment offset. In the simplest approach, the collection of durable objects could be thought of as one 8088 segment and object pointers in the data represented as byte offsets within the file (segment).

As implemented, each object instance could contain a series of arbitrary scalar quantities, or pointers to other instances of the same class. A more heterogenous data structure -- with objects of one class owning collections of another -- would require a universal protocal for identifying the disk-resident object classes (segments). For simpilicity's sake, the segment number for each file of a user's programming environment could be uniquely assigned by a segment manager.

A collection of durable objects could then be arranged in the same way as non-durable objects, either using keys, or ranking, some other form of ordering, or unordered lists. Such a collection would be a database, but manipulated via customary object-oriented protocols.

Independently of this proposal, SIMSCRIPT creator Harry Markowitz has designed and implemented the EAS-E database language **[Markowitz 1984]**. The language is based on the SIMSCRIPT concept of entities, attributes and sets ("objects, properties, and collections). The same protocols can be used to manipulate both memory-resident and disk-resident entities. The EAS-E compiler is based on an early SIMSCRIPT II compiler, but is now written in EAS-E, allowing it to manipulate EAS-E memory and disk objects.

The EAS-E system, like the Smalltalk-80 environment, includes a browser that allows the user to view and modify the data structures. Such an object editor would allow the LCS user to access and change individual database records (object instances) in much the same way as standard microcomputer-based database programs do. The editor would be far simpler in implementation, however, because it is used only for query and update purposes, not complex reports.

4.10 Other Structured Programming Constructs

The existing SIMSCRIPT II.5 contains many of the features of a structured programming language in addition to its unique high-level language constructs that directly support simulation.

As noted, SIMSCRIPT provides for recursive procedures and dynamic data structures, as well as standard simulation constructs (events, processes, resources).

The differences between the control structures of SIMSCRIPT and other structured languages are more ones of form rather than function, so it is proposed that LCS adopt those of the current SIMSCRIPT II.5 language. The simplest such structure is the **if** statement:

```
if X < 20
     ''true case
else
     ''false case
endif
```

The other primary (non-iterative) control structure of SIMSCRIPT II.5 is the case-selector block, recently added to the current definition of the language **[West 1985]**. The **select case** statement allows selection of one of several alternatives, based on a numeric or string expression:

```
select case STRING
     case "YES"
          '' one possible case
     case "NO", "NEIN", "NYET"
          '' duplicated case
     case "0" to "99999"
          '' range of cases
     default
          ''none of the above
endselect
```

For looping, a series of repeated actions may be performed by one of the following iterative groups:

```
while BALANCE <> 0
     do
     ''block of iterative statements
     loop

for COUNTER = LOWER to UPPER
     do
     ''block of iterative statements
     loop

for each OBJECT
     do
     ''block of iterative statements
     loop

for each OBJECT in COLLECTION
     do
     ''block of iterative statements
     loop
```

Any of the loops may include one of the statements:

```
leave     ''exit the loop

cycle     ''continue with the next iteration
```

Object-Oriented Distributed Simulation

From a conceptual standpoint, providing both embedded record structures and multiple inheritance are redundant and add unnecessary conceptual complexity. In addition, such embedded structures pose a serious practical problem in that they make it difficult to perform stringent run-time object validation, including dynamic type checking. Such problems are not found in the previously stated "pure" forms of object structures, including multiple-path class inheritance, instance-oriented inheritance, or by using attributes that are object pointers.

The impact of modules (Modula-2) or packages (Ada) on LCS has already been addressed in section 4.9. Such an instance-based modular program is necessary if the simulation data is to be adequately distributed across a parallel-processing system.

4.11 Deferred Language Issues

The syntax of a few remaining constructs, while straightforward to design, has not yet been tackled. Such constructs would be arbitrary in specification, but necessary for a complete implementation.

These would include:

* Derived data types
* Range and subrange limits
* Enumerated constants

The existing SIMSCRIPT syntax is also repetitive when defining attributes of entities, in that each attribute must be declared twice: once for the inclusion in the entity, the second to set its scalar type. One possible solution would be a Pascal-like syntax of

```
every PLANE has
     WEIGHT : integer,
     SPEED  : real,
     NAME : text
```

A provision is needed for symbolic constants and inline-procedure expansion. The SIMSCRIPT language provides the first using the **DEFINE ... TO MEAN** construct, while both C using a macro preprocessor. In contrast, Pascal provides a **CONST** identifier for the former, and some implementations have a special type of procedure defined as **INLINE** to implement the latter.

Finally, a way is needed to define interfaces between a user's program and a standard library package, typically an LCS environment. The Modula-2 **import** and Ada **packages** provide a way of manually referencing such definitions in the routines that use them. A more powerful, but complex alternative is to use a programming environment -- such as that found in Smalltalk-80 -- to maintain the interfaces to the standard library. The use of SIMLAB environment for PC SIMSCRIPT II.5 **[Mullarney 1984]** would provide a starting point for development of such system for LCS.

CHAPTER 5: DISTRIBUTED SIMULATION USING LCS

One of the design objectives discussed in Chapter 2 was for a language that would be appropriate for single or multiple processors. For both practical and conceptual considerations, the implementation details should be as similar as possible in order the minimize the implementation dependencies that could diminish portability between single and parallel systems. One such issue is how objects are referenced on local and distant CPU's.

When a model is running as a distributed simulation, the basic requirements are threefold:

* Distribute the data
* Distribute the computation
* Synchronize data and computation

Various approaches to these requirements were discussed in Chapter 2, with particular emphasis on the issue of computation synchronization.

Even if these general parallel-processing requirements are met, a simulation may experience yet another problem -- the issue of non-determinism. Failure to achieve reproducible results -- even if the differences are minor -- will likely lead to user rejection of the simulation approach.

5.1 Parallel vs. Sequential Execution

As noted earlier, a pure object-oriented system running on an MIMD system would use the object-message paradigm to implement parallel processing. Low-level operating system utilities are called to copy a message list from the sending object's processor to that of the recipient object, and then place the message on the receiving object's pending list.

On the other hand, a single-CPU compiled object-oriented language -- such as Simula or Object Pascal -- uses subroutine calls by the sending object to the method routine of the recipient object. Direct calls can be used where static typing is available, while an indirect table lookup is required where the object is dynamically typed.

From a user standpoint, the differences between the two implementations of message passing should be transparent. While a inter-processor message is slower than an intra-processor message, the behavior and side effects must be identical. This allows a smooth transition for a model moving from a single processor to a 1,000-node system.

In some distributed simulations the number of objects may greatly exceed the number of processors. For ease of moving existing models, as well as performance considerations, it will often be desireable to group a tightly-coupled set of object instances. The LCS environment provides a conceptual framework for specifying such groupings.

The decision can be made at compile or link time that each instance of a class of parent object and its children should be clustered for sequential computation on a single processor. The interactions within the environment would be reduced to the simple sequential ones from the more complex parallel ones, improving both the speed and determinism (Section 5.6) of the simulation.

Such a single-CPU grouping of objects is referred to as a <u>local environment</u>. It would appear reasonable to make the distinction on a class rather than instance basis, so that all instances of the environment (parent object) would be bound to a single-CPU.

Not all environments need be local, however. In a simulation with a large number of processors and relatively few objects, it may be advantageous to separate the child from the parent objects. When the child objects are not bound to the CPU of the parent object, the term <u>distributed environment</u> is used. If there are multiple layers of environments, some may be local while the parent environments are global.

The assignment of an environment class as local or distributed would be most easily implemented at compile time. However, the time necessary to recompile a large simulation -- merely to run a data case with a differring degree of concurrency -- would imply that a link-binding approach is preferrable.

One way to implement this would be to bias towards the most optimistic case -- a local message -- which is also where additional message-passing inefficiency would be most noticeable. All message-passing could call the corresponding method routine for the receiving object's class. In a distributed environment, the method would send an explicit mesage using the appropriate system subroutine.

A single-CPU system is, of course, a degenerate case, in which the entire program is contained within a local environment. This mechanism could be used for all message passing, allowing easy migration from the most sequential to most parallel of systems.

5.2 Referencing Distributed Objects

In single-processor structured languages, objects are traditionally referenced using direct hardware (virtual) memory addresses. This approach will also work with shared and quasi-shared memory distributed systems with a common address space, such as the Butterfly.

However, a different approach is required when object instances will communicate on MIMD computers using inter-processor messages. There is no direct access to the objects on the remote processors, so a protocol must be provided for finding and addressing an object at an arbitrary location in the system.

For the general case of distributed computation, the object reference variable cannot just be the memory address as in the single-CPU case. Instead, some form of logical pointer is required. While the usage of the logical pointer may vary between a local and remote memory access, the representation of the pointer should be similar. so that the two may be used interchangeably in utility routines, etc.

The decoding of a logical pointer may require a world map (as proposed by **[Jefferson 1984]**) on each processor to decode an object reference. The map would be indexed or hashed on the logical pointer, and would normally include information such as the processor that the object is now on.

The need for a world map is increased if the system performs dynamic relocation, in which the operating system attempts to balance the load across the various processors. When dynamic rellocation is allowed, objects cannot no longer assume that the receiving object is at the same CPU as when previously used. In particular, a intra-processor memory access can become an inter-processor message transmission at any time.

Dynamic relocation has performance penalties: Each relocation requires a sizeable overhead, and relocation that breaks a local environment cluster should probably be avoided. However, dynamic relocation may be essential in many cases, since even the most clever of static allocations may not assure adequate utilization of a large number of processors in a complex simulation.

Given these parameters, what representations of the logical pointer are possible? The options are varied, but they include:

Character string. The prototype implementation of Time Warp ([**Beckman 1984**]) uses a character string to uniquely identify each instance in the system. The text includes the object class name and a number. This choice is good for debugging but entails serious performance penalties, particularly when used within a local environment.

Numeric Indexes. A combination of various numeric indexes could be used for efficient table lookups. Certain minimum ranges must be considered for large models:

Number of processor nodes > 256
Number of object classes > 256
Number of object instances > 65,536

This suggests that a unique 64-bit pointer could be built with the processor number, object class and instance of object on the processor, as follows:

```
1              16  17              32
 _______________________________________
|      CPU      |      class          |
|---------------+---------------------|
|        instance number              |
 _______________________________________
```

However, the experience of systems built around such implementation-dependent limits suggests that even these apparently generous values may prove inadequate in the long run.[18] In addition, data structures and argument lists are typically built around the assumption of a 32-bit parameter, whether by reference or by value.

Handle. The term object handle can be used to refer to a second-level indirected pointer. In a single-processor system, the higher-level language will use a handle, which contains a pointer to a hardware memory-pointer. If the memory pointers are stored sequentially in a block of known length, this permits dynamic memory relocation of objects within a processor's address space. This could be used in a single-processor case as follows:

```
handle ----> |  pointer  | -----> |          |
                                   |  object  |
                                   |    ...   |
                                   |__________|
```

For a reference to an object on a remote system, the pointer is set to an arbitrary invalid value. The remaining data in the object referenced by the pointer indicates the object's location in the same way as with the numeric values.

```
handle ----> |  INVALID  |
             |  CPU no.  |
             |instance # |
```

As with memory relocation, the use of handles to entries in the world map facilitates the dynamic relocation of objects, since updates in the world map on each processor will automatically be reflected in any future access to a relocated object.

The pointer object could be used to store additional information relevant to the object, and, in particular, the object's class value or template pointer, which will be frequently used in method dispatching and attribute offset calculations. It could also be used for dissimilar object addresses, such as durable objects.

The handle approach could be further enhanced if the memory manager would detect the particular "remote object" invalid address and automatically dispatch the appropriate message for an instance variable reference. Such a memory manager would begin to resemble the node controller of the Butterfly Muliprocessor (see Chapter 2).

However, the choice of approach will strongly be influenced by the particular distributed simulation paradigm used and the characteristics of the hardware and operating system. In a shared or quasi-shared memory machine, a simple segmented address could be used, although more than 32 bits would be required for typical systems.

5.3 Distributing Data

In an object-oriented simulation, the state of the simulation is defined by the data in the objects. In a closed object-oriented implementation, there are no passive objects: state values can be obtained only by asking an object for the value by sending a message.

This restriction is unduly harsh when considering an open implementation of a compiled, object-oriented language such as LCS. It may be desireable to allow direct access to the memory locations when on the same CPU, thus making the issue of assigning data to each processor one of crucial importance in realizing improved throughput.

5.3.1 Global Data

Existing global address spaces encourage programmers to base their large programs around globally-accessible data structures. Such data can be easily accessed by any routine at any time with no performance penalty.

For concurrent simulation to be effective in a non-shared memory machine, this data must be divided up across the various processors -- in parallel with the division of the corresponding computational tasks.

The first temptation is to keep the existing address framework, but make every piece of data an "object" and each data access a "message." This is a conceptually pure approach, but it quickly becomes apparent that the system will become overloaded with messages unless some intelligence is applied to the problem of grouping the data.

For example, consider a linked list represented using conventional single-processor techniques. Take the SIMSCRIPT II.5 fragment:

```
for each PLANE in FLEET with PLANE:TYPE = "747"
     add PLANE:LIFEJACKET.COUNT to TOTAL
```

Now let's look at the low-level psuedocode expansion of this loop. To remind us of the penalty associated with each message transaction, the messages interactions will be shown as **ask** statements, even if the "colon notation" would be more syntactically correct.[19]

```
     PLANE = F.FLEET
'LOOP'
     if PLANE <> null
          ask PLANE TYPE yielding TMPSTR
          if TMPSTR = "747"
               ask PLANE LIFEJACKET.COUNT yielding TMP
               TOTAL = TOTAL + TMP
          endif
          ask PLANE S.FLEET yielding PLANE
          go LOOP
     endif
```

As you can see, each repition of this very simple loop will require three messages (TYPE, LIFEJACKET.COUNT, S.FLEET) -- messages which may require relaying through a number of processors. If the system is running with asynchronous time, each sending/receiving object pair must be synchronized (section 5.3) before the data is accessed.[20] A simple operation on a single CPU has become one creating hundreds of inter-processor messages.

A number of alternatives exist for this and other cases

Cluster objects on one CPU. When the grouping of objects is tightly coupled and such loops are common, it makes sense to group the objects on one processor, such as through the LCS environment.

Use a collection appropriate for distributed data. Obviously, the standard single-CPU linked list shown is not appropriate for a MIMD system lacking shared memory. Alternate approaches would reduce the message traffic while allowing the data to be distributed. This topic will be examined later in this chapter.

Defer all calculations to the object. This approach would be quite appropriate for a language such a ROSS, but could be developed with extra programming effort in more conventional object-oriented languages. In this case, it could be conceptualized through the psuedo-code formulation

```
tell each AIRPLANE
     (if TYPE = "747"
          tell OWNER JACKETS(LIFEJACKET.COUNT)
      endif)

method JACKETS(COUNT) for ME
   add COUNT to ME:TOTAL
end
```

The first approach is the only one with a speed comparable to the single-CPU case, but it is applicable only to a certain very limited class of problems. The second and third are alternatives when dealing with objects that must be distributed across various CPU's. All three approaches have their application, and all may be necessary in order to reduce message traffic to an acceptable level.

If the data is assumed to be part of an object instance, and can be used freely (locally) by that instance, then the problem focuses on the remaining global accesses outside that instance or environment. Most classes of global objects (see Section 4.8.1) will be referenced by other objects distributed across a number of processors.

In analyzing an existing model to distribute the globally shared data, the potential message traffic and associated delays must be taken into account before deciding how each class of data is to be treated.

A decision matrix can be constructed based on the frequency of access to the object's data (relative to other computations) for both query and modification. Such a matrix is shown in Figure 5-1.

		USED OFTEN	
		NO	YES
CHANGED OFTEN	NO	either approach-->	distribute data across processors update via broadcast messages
	YES	centralize data on one processor access via query messages	redesign data relationships

Figure 5-1: Alternatives for treating global data

The case of data that is rarely used at all is the easiest case, since very little performance impact is found no matter what the approach. This case lends itself to the least-effort approach, which may involve adopting the scheme used for a related class of objects.

If the data is frequently modified (either globally or locally), but infrequently accessed by other objects, then the conventional object-message approach is appropriate. This is a clean, standard object-oriented interface.

If the data is static or quasi-static, it should be replicated across all CPU's where it is used. If a change is made in the data, a message is broadcast to all replications of the data indicating the new value. Terrain data and communications networks lend themselves nicely to this approach; another example is the world map of object locations.

If the data is frequently modified AND frequently accessed, neither approach will produce adequate performance in a many-processor system. An example would be the topology of a rapidly-changing communications network. In this case, the network would have to be localized by some criterion, such as the object's role in the system hierarchy, its location in the physical system, or both.

For example, objects could be broken up into regions. Objects would deal regularly only with the network in their immediate region, and any extra-region accesses would be avoided wherever possible. If the objects (say, messages) in the region spanned several processors, then either approach could be used within the region. A geographic region could be implemented as an environment.

As with environment partitioning, the replication of global data across multiple processors should be transparent to the programmer using that data. This could be through a preamble declaration of "distributed data" or by providing a standard library of inherited behaviors for accepting and broadcasting attribute changes.

5.3.2 Distributed Collections of Objects

Taking a look at the previous example, it seems apparent that a more general approach needs to be taken in maintaining collections of objects in a parallel-processing system.

In particular, the housekeeping associated with maintaining the collection needs to be separated from the properties of the objects in the collection. This housekeeping is (or should be) tightly associated with the object that owns the collection, including keeping the housekeeping data in the same environment as the owner object.

For example, Smalltalk-80 defines the class **Link** for minor housekeeping objects associated with **LinkedList** collections. Each type of **LinkedList** has its own corresponding type of **Link** that is used to maintain the collection. From an implementation standpoint, each member is added to the list by creating a new link, adding it to the owner's linked-list, and placing a reference to the member object in the link. The maintenance of the **Link** entries is comparable to the existing practice of SIMSCRIPT (and other languages) for maintaining the member entries. An example of this is shown by the Figure 5-2.

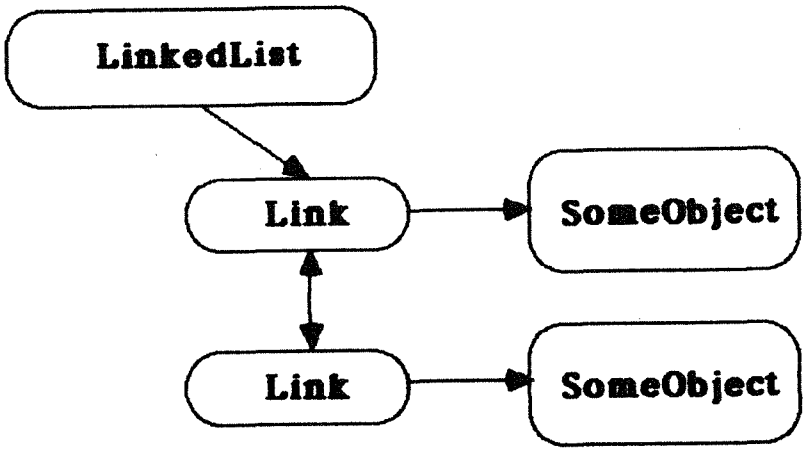


Figure 5-2: A Smalltalk LinkedList

In a distributed simulation, the **Link** objects should remain on the same CPU as the owning object, which is responsible for adding, deleting, and searching through entries in the collection. This use of a placeholder **Link** (or equivalent) would be necessary for all collections in a distributed simulation, except those contained entirely within a local environment.

The organization of a collection has important implications on its use in a distributed system. All existing SIMSCRIPT collections are ordered, whether that property is required or not; many would be more appropriately treated as unordered or keyed.

Ordered collections face issues of non-determinism when it comes to establishing an order in a parallel-processing environment. In the absence of some unique and non-reproducible ranking key, all ordered collections -- even ranked ones -- must default to some sort of FIFO or LIFO discipline to establish an order between two similar member objects. Messages filing objects in a collection will be time-stamped, of course, and that time would normally be used as a secondary ranking criterion.

Many collections do not require an explicit order and could just as meaningfully be used as unordered collections. Even larger subclasses of problems require an explicit sort key that can be used to access an object (or group of objects) that have a particular attribute or attributes. For effective use in distributed systems, the key should be included as part of the **Link** maintained on the owner object's CPU.

As noted in the earlier example, many operations on objects in collections could and should be performed in parallel. But this

will not usually be possible with an ordered collection, which implies that actions on the member objects will normally be done sequentially. An example of this would be a waiting line at a restaurant, in which the host finds the first party of 4 in line when a four-person table opens up.

However, sequential operations on unordered or keyed collections will produce random results and are unlikely to be used. Instead, the operations will typically involve all the member objects, possibly selecting only a subgroup, as in the example above. Such operations could be performed in parallel without concern over possible unusual side-effects.

5.4 Distributing Computations in a Simulation

Many of the object classes alluded to are considered "passive;" that is, the object exists for storing a data state and not to perform some active role in the simulation.

However, the bulk of the computation in the model will generally be associated with "active" objects. The distribution of these active objects plays an even more fundamental role in maximizing the throughput of an MIMD machine, since the distribution of the instances of the active objects will also distribute the computations associated with those objects.

If the operating system supports dynamic relocation, then the active objects can be freely juggled among available processors, much as task are assigned to memory in a time-shared virtual memory system.

While dynamic relocation offers the potential for maximum utilization, initial efforts should focus on solutions that do not require this, much as task swapping systems were implemented before page-by-page virtual memory systems. The best (in fact, the only) clue for making static instance assignment comes at the time of instance creation.

In a strictly hierarchical model, the creation of a new object can be taken to indicate an object that will be frequently used by the creator. Thus, the top-level nodes of the hierarchy should be dispersed throughout the system, and, in turn, the lower level nodes allocated across adjacent processors.

Most models do not have such a pure hierarchy. A common example is the process generator: the object **GENERATOR** may create thousands of **TELLER** instances and, once they are created, have no further interactions.

However, the LCS environment approach provides a conceptual framework for such instantiation preferences -- particularly since the child-objects in the environment are accessible only within that environment and may communicate using attributes of the parent object.

Within a local environment, of course, new instances must be created on the same processor. It seems reasonable to allow the user to make such an explicit assignment, even at the risk of critical-path bottlenecks later on. As the only penalty is speed, and most simulations are run many times to gain statistical validity, such a poor human decision can be reversed on later runs if detected by standard performance analysis tools.

5.5 Non-Determinism in Distributed Simulation

For models that are executed sequentially on a single CPU, modern simulation languages provide an explicit order of execution for each event, complete with class- and instance-oriented tie-breaking rules.

Even absent tie-breaking rules, single-CPU models can use a final tie-breaking algorithm: first in, first out. This means that if event **A** schedules an **A'** for time **T**, and **B** schedules a **B'** for the same time, the order of (**A'**,**B'**) will be unambiguously determined by which of the two earlier events was executed "first" on the processor.

Network solutions assume a well-defined sequential ordering of interactions along each path betwen an object and other objects. With explicit ranking of the priority of each path approaching the object, the results would then be completely deterministic and reproducible.

However, more general approaches to distributed simulation -- notably Time Warp -- do not provide such an explicit ordering criterion. The only alternative to non-determinism is by requiring explicit tie-breaking rules for ordering all message arrivals, with each object establishing its own protocol for ordering messages with simultaneous time stamps. Such an ordering would then tend to cause more frequent rollbacks in a Time Warp system.

One way to reduce the incidence of non-determinstic "ties" would be to introduce psuedo-random noise into the arrival time of each message as it is sent. If the message times are non-identical and reproducible, then there will be no ties that need to be

broken on the basis of which sender completed its computations first.

Unfortunately, psuedo-random number streams themselves would become non-deterministic themselves unless maintained on a master "random generator" object with a deterministic ranking on sample requests. Such streams are the primary source of stochastic behavior in Monte Carlo simulations. The errors introduced by non-determinism are multiplied and may grow without damping, since the values of successive samples will vary widely.

Large simulations have been built without any reliance on psuedo-random values, and such simulations would work well on a parallel system. However, further research is needed into psuedo-random generators that do not rely on successive sequential sampling to generate randomness.

5.6 Interfaces to Time Warp

While the issues discussed earlier in this chapter would apply to most, if not all, distributed simulation paradigms, the Time Warp operating system imposes a number of particular requirements upon a model running under it.

The current prototype implementation of Time Warp is based on a small number of operating system entry points. These entry points have been used to construct a simple model to demonstrate the use of the Time Warp operating system.

However, a more general abstraction is needed in the interface between a simulation modeler and the operating system, much as a Pascal programmer does not have to worry about coding the record-buffering logic when writing a **writeln()** statement. This abstraction should be provided by both the lower operating system layer, and the simulation language system that rests upon it.

In invoking Time Warp system services to send a message from one object to another, the following information must be provided by the sending object:

* Logical pointer to recipient object
* Delivery time
* Message selector
* Message arguments

In addition, a distinction needs to be made between synchronous and asynchronous transmissions of messages. This could be part of the Time Warp protocol, or implicitly layered on top of it.

At the receiving end, each object should specify a special entry point to explicitly rank incoming messages. Such a behavior, invoked by Time Warp when sorting messages with identical times, allows the individual object to order messages by selector priority or argument values. If used for breaking all such ties, this method routine would greatly reduce the amount of non-determinism in a distributed Time Warp simulation.

Each method of the recipient object must also designate whether execution of a particular method will cause a state rollback for the object. This could be detected by the compiler or explicity declared by the user, and implemented by providing Time Warp an **ISEVENT(selector)** Boolean function for each object.

The current implementation of Time Warp assumes that the state of an object is defined by a series of sequential memory locations arranged as a C structure. This precludes more realistic structured data constructs, as well as the particular requirements of an object-oriented simulation language. For example, if a Time Warp task is actually local environment of several objects, the state of the task includes all the instances of objects within that environment.

A more general state-saving solution could involve a <u>state identification template</u>, which would indicate the format and location of the various components of the state of the object, including:

* Object attributes
* Dynamic strings
* Recursive variables
* Child objects
* Arrays
* Compound structures, such as linked lists

The approach for specifying a language-independent state identification parallels that for a high-level symbolic debugger. The Common Object File Format of UNIX System V **[ATT 1984]** is one example of such an approach. The template could, of course, be used to implement a run-time diagnostic and debugger system.

An alternative to the use of a state identification template would be a replication entry point for each task, which would place the burden of producing an identical state value upon the user or his/her support language. This would be the most flexible approach, but would result in a great deal of wasted effort if more than one language were used for simulation.

CHAPTER 6: RECOMMENDATIONS AND CONCLUSIONS

The preceding sections have outlined one possible solution to the problem of developing large military simulation in a distributed processing environment.

Other solutions are possible, of course. Some believe that the development of object-oriented and simulation capabilities should be made upon one of the modern structured language of the Pascal family.

Others believe that simulations based upon Lisp offer great promise, especially in models with strong artificial intelligence components. However, there are no immediate prospects for parallel dedicated Lisp machines, or indications that the language would handle parallel processing well.

6.1 Hardware and Software Recommendations

6.1.1 Hardware Systems

For discrete simulation, future emphasis should be placed on multiple instruction stream/multiple data stream homogeneous systems. An effort should be made to identify the ratio of simulated objects to system nodes that will produce the most cost-effective use of the computer hardware. It seems unlikely that this ratio will found to be less than 2:1: a number closer to 20:1 may be appropriate.

Given current models with 200 to 2,000 objects, it would seem that a 100-processor system would be more appropriate for initial evaluation than a 1,000-processor system. Also, given the computation and memory requirements of large simulations, resources should be devoted to assuring a minimum performance at each node roughly equivalent to a VAX-11/780, rather than a larger number of PC-class nodes.

At the same time, it is too early to say which of the various approaches to parallel computation will prove to be significant in the long run. The technology for no-shared-memory MIMD hardware is furthest developed, but places the greatest burden on finding a viable software approach to parallel processing.

Object-Oriented Distributed Simulation

Shared and quasi-shared memory machines, such as the Butterfly and UltraComputer, have the greatest technical obstacles to overcome. But the provision of shared memory would minimize the software changes necessary from a single-CPU approach and reduce the problems of synchronization and non-determinism found in the no-shared-memory case.

Finally, the small-grain Dataflow architecture offers a formal hardware and software solution to the problems of synchronization and non-determinism, while extracting the maximum concurrency from a problem. As yet undetermined is whether the penalties associated with the architecture exceed the concurrency gain, and whether Dataflow is a relevant methodology for expressing major discrete simulation models.

6.1.2 Distributed Simulation Paradigms

The author believes that no feasible alternatives to use of the Time Warp operating system have yet been developed. It is closer to implementation and addresses the issues of automatic concurrency and synchronization better than any identified alternative for distributed simulation.

The current Time Warp implementation is proceeding towards a multi-CPU implementation that will do much towards answering practical questions regarding its efficiency and productivity. The implementation will also serve to refine the Time Warp concepts through actual use.

However, before attempting to develop major simulation models using Time Warp, the following issues must be addressed:

1) The prototype implementation of Time Warp provides only limited support for inter-task message passing, requiring that the user understand low-level Time Warp implementation characteristics when building a simulation model. Higher-level tools need to be provided -- either through the operating system or a higher-level language -- to abstract these characteristics to more general properties not peculiar to Time Warp, such as message synchronization, side-effects, return values, and sequential ranking of incoming messages.

2) The current implementation makes unrealistic arbitrary assumptions about the simplicity of a simulation model and its data structures. As noted in the previous chapter, a more general approach is needed to the definition of the state of an object.

3) The Time Warp operating system should include a monitor for analysis of object-object communications flow, towards improvements in task-CPU assignment and Time Warp control parameters.

4) As one alternative to dynamic relocation, the operating system development team should expore the use of use of compiler or user clues in the instantiation of new objects. In certain cases -- such as the LCS environment approach described in Chapter 4 -- such clues could offer a far less complex and far more effective alternative to the sizable performance penalties associated with the dynamic relocation capability.

5) Time Warp implementors should evaluate the desirability of allowing users to "bundle" tightly coupled objects -- such as a tank and its driver -- to allow significant performance optimizations based on an assumption of same-CPU assignment.

6) A series of performance-analysis tools should be developed to measure the efficiency of the operating system and applications running on it. Such tools could include critical path analysis **[Berry 1985]**, and a breakdown of CPU usage into the categories of message wait, system overhead, checkpoint/rollback, useful work and unused time.

6.1.3 Simulation Language

As noted earlier, the trends in current MIMD hardware and distributed simulation paradigms suggest that the object-message will be a part of future distributed simulations.

This report proposes one such language, which specifically addresses the problems of parallel-processing, as well as including integrated simulation tools that have been proven in two decades of building major discrete-event models.

Other approaches would be feasible, particularly if they were to build on an existing object-oriented language. A compiled language should be used if speed is not to be sacrificed in favor of development flexibility.

The language Simula67 would appear to be an obvious choice, as it is an existing, compiled object-oriented simulation language. While it has been somewhat successful in Europe, it has traditionally lacked in the United States all the necessary components for a successful software product -- including quality

documentation, user training, and professional support.

One of the important factors in Simula's reception in the U.S. market has been its basis on Algol-60 which, with a few exceptions, has been little used here. For whatever obscure cultural or historical reasons, Algol never caught on during the 1960s and early 1970s, and its software niche has since been taken over by one of its children, Pascal. This structured programming niche will probably be inherited by one of its grandchildren, such as Ada or Modula-2.

As such, the Object Pascal language uses a newer and more standard basis for a compiled object-oriented language, and the principles therein could be used for either an Object Modula-2 or Object Ada.[21] The Object Pascal language is conceptually clear and complete. On the minus side, the relative youth of the language leaves it unclear as to what practical impact it will have on the programming world. Also, the flexibility of its object orientation is severely limited when compared to Smalltalk or Flavors, although this may be a slightly unfair comparison.

The language C++ offers, in the author's opinion, a more elegant solution to the problem of a compiled object-oriented language. However, its owners have thus far shown little inclination to market it as a commercial product and may never do so. In addition, those who are not fans of the parent language C will find that it retains many of the structured programming deficiencies of its parent.

For either Object Pascal or C++, a complete simulation support system would also have to be developed to provide both compile- and run-time tools for the development of large models. The design of such extensions could be based on one of the existing procedural simulation languages, such as Simula or SIMSCRIPT. The number of man-years required for such a solution would depend on the completeness of the solution, although the use of an extensible object-oriented base language could be expected to shorten the process considerably.

6.2 Recommendations for Further Directed Research

A new generation of simulation tools for use in either a single- or multi-processor environment should be developed in three phases:

Phase 1: Language Design and Selection

Identify a completed design for a simulation language appropriate for distributed simulation. One such design would be the Language for Concurrent Simulation described herein. Further research would be needed to develop a design based on the alternative languages discussed in the preceding section.

Phase 2: Implement A Single-CPU Prototype

(a) Implement the distributed simulation language on a conventional single-CPU system, such as being done with the Time Warp operating system.

(b) Adapt an existing combat model -- or a significant fragment thereof -- to use this prototype simulation language. For this task to be successful, the model should be medium-sized (5,000-20,000 lines) and the conversion should be done by experienced simulation analysts with a proven track record in military models.

For example, an AMIP model could be identified and the Army team that developed it could port it to the new system. This development should be in close consulation with an implementor or instructor from the group implementing (a).

(c) Run the newly-ported model in a single-CPU environment using a parallel-processing performance analyzer such as the Time Warp simulator. A critical path analysis (such as described in **[Berry 1985]**) performed

(d) Evaluate the results of the model development and use. Is the simulation methodology natural? Is it flexible enough to develop major simulations? Does it allow expression of adequate parallelism? What could the conversion costs be expected to be for existing models?

Phase 3: Parallel-processing Implementation

(a) Any simulation language(s) validated in Phase 2 should be implemented on the appropriate hardware and operating system configuration. The initial implementation should be on a MIMD system with a moderate number of nodes (32 to 256).

Should current evaluations prove favorable, it is anticipated that this initial test system would be Time Warp on a Hypercube-type computer.

(b) The model developed in Phase 2 should be modified to increase the identified parallelism. Modifications should also be made to meet the requirements of the particular hardware/operating system configuration.

(c) When the preceding tasks are completed, accurate measurements would be obtained under the widest possible range of conditions. These tests should include large and small data scenarios, larger and smaller hardware configurations (by disabling processors, if necessary) and alternate assumptions regarding object coupling and object-processor assignment algorithms.

The results of these measurements should attempt to identify a method of estimating the processor utilization curve for a given hardware/data combination, and the make predictions as to the proper system configurations for the cost-effective use of parallel-processing.

(d) Evaluate the feasibility of all three simulation components -- language, operating system, and hardware -- for use in production simulation applications. Make recommendations for further modifications or use of these components.

It is the author's opinion that a simulation language can be developed for running major military models in a distributed environment. It is recommended further research towards running such models be funded using the three-phase approach outlined above.

NOTES

1. Personal communication, Dave Toved, Motorola Seminconductor Inc., March 21, 1985.

2. Personal communication, Dave Mankins and John Goodhue, Bolt, Beranek & Newman, March 20, 1985.

3. Personal communication, Jim Blossom, Los Alamos National Laboratories, March 28, 1985.

4. At first glance, it might appear easiest to map each "process" of a user's simulation to a "process" of the Time Warp operating system. A careful examination, however, suggests that this usually will not be the case, as discussed further in Chapter 5.

5. This approach is an exact parallel to the UNIX concept of a "pipeline", which has been shown through extensive use to work quite reliably and effectively in coordinating multiple sequential tasks on a single-CPU system. The finite limit is typically 10,240 bytes, or about three pages of solid text.

6. "How to Write User-friendly Software," lecture by Larry Tesler of Apple Computer, Inc., MacWorld Exposition, San Francisco, California, Feburary 23, 1985.

7. Sometimes this is implemented literally, by partitioning the machine's actual address space into regions occupied exclusively by objects of the same type. More often, however, the machine's natural addressing is augmented by type bits, or else unused low-order address bits are used for type information, since usually the smallest-sized object is much larger than the machine's addressing granularity.

8. At first glance, it would seem necessary to restrict the hierarchy of clases to acyclical directed graphs, to avoid recursive definitions. As it turns out, it is possible -- and even desirable -- to relax this restriction.

9. If the user is not allowed direct access to attributes, there is no way for him to create attribute accessor methods.

10. Although the terms "instance-based inheritance" and "message forwarding" refer to two totally different concepts, instance-based inheritance will be shown in practice to be a special case of message forwarding.

11. I am grateful to Dr. Alasdar Mullarney for first making this suggestion, and for fighting the temptation to use the delimiter in SIMSCRIPT II.5 for more gratituous purposes.

12. This could perhaps be more clearly expressed as

PLANE:ENGINES(ENGINE.NO):RPM

where **ENGINES** is an array of pointers to **ENGINE** objects. The syntax is less ambiguous than the combination of left- and right-associative operators in

PLANE:RPM(ENGINE.NO)

It is also more cumbersome.

13. The compiler could attempt to automatically detect the first usage of the value and enforce the synchronization at that point, but would introduce a great deal of conceptual and implementation complexity for what is likely to offer only a marginal benefit.

14. The use of postional parameters is a tempting alternative. This is produces a less readable result, but is more compact. More signficantly, it provides much greater flexibility, as demonstrated by the constructor facility of C++ **[Stroustrup 1984a]**. The list of values then becomes an argument list to the initialization method, which can decide their meaning as it chooses.

An added benefit is that the argument list syntax parallels that of the **activate** statement of SIMSCRIPT II.5.

15. For strict genetic accuracy, the terms "twins" and "siblings" or "siblings" and "stepsiblings" might be more appropriate, but this would be a lot more confusing.

16. The order might in fact, be reversed, if each instance of **PARENT** were executed asynchronously in differing cpu's.

17. It is tempting not to require explicit object pointers for changing private attributes of the parent object, since they are like the local variables of an Algol outer block or a Modula-2 module. Strict consistency between public and private attributes, however, would require that they be used, as in

AIRPLANE:AUTO.SET.

18. Common examples of painful transitions due to memory architecture addressing limits include Univac Exec 8 and Honeywell GCOS-8 (262k words), IBM's MVS (16 mb), and the as yet unresolved issue of IBM's PC-DOS (640kb). To ignore the history of such design failures is to be doomed to repeat them.

19. It has been proposed that users be forced to explicity acknowledge each use of message-passing to make sure they are aware of its performance cost and thus use it sparingly. This computer science purism is analogous to asking each user to program in machine code to make sure (s)he knows each byte that is required by the program!

The whole history of modern languages has been to provide more powerful tools to users, and then caution them when a great deal of user convenience is exacted at the expense of performance. Users of Lisp and APL can attest that the productivity gain for certain classes of problems is worth whatever the cost in computer resources.

Discipline alone will not solve the problem of building effective concurrent simulations. The appropriate tools and proper training will, in the author's opinion.

20. The implementation of the Time Warp operating system **[Beckman 1984]** has proposed that this restriction be relaxed to enforce only a one-way synchronization. If the receiving object is ahead of the sender in simulated time, it will go to one of its snapshots of its previous states (saved for rollback purposes) to satisfy the query. The granularity of the snapshot approach (to save memory) may require some degree of re-execution.

This suboptimization has a potential for significant performance improvement, particularly in avoiding deadlock situations. But, from a language design standpoint, it can be ignored, since it offers at most a 50% reduction in synchronization requirements.

21. To conform to trademark regulations by the Ada Joint Program Office, use of the name "Ada" for an Ada-like superset would require a frequent disclaimers as to its non-conformance to ANSI/MIL-STD-1815A. Another name might thus be more appropriate.

GLOSSARY

The number listed in parentheses represents the section where term was first introduced.

antimessage (2.2.1) a **Time Warp** message that is used during **rollback** to cancel a message sent earlier in the simulation.

around method (3.4) an approach to **method combination.** An around method is given complete control when the corresponding **message** is received, and may choose which, if any, of the inherited methods to invoke.

asynchronous message (4.5.1) a message that allows the sending object to continue before the receiving object completes its corresponding **method.**

asynchronous time (2.2) an approach to distributed simulation in which each object may have its own value for simulated time.

attribute (3.2) a variable associated with each **instance** of an object.

browser (4.9) an editor for **durable objects,** usually with a visual screen-oriented interface.

child object (4.8.1) an object that is declared within the scope of another object, which is referred to as its **parent object.**

class (3.2) a group of related **objects** that share similar properties, but may have different values

class-based inheritance (3.5) an inheritance in which all **instances** of a **subclass** inherit one or more properties from its **superclass.**

class variable (4.7.1) a value associated with all instances of a class of objects.

closed implementation (3.1) refering to **object-oriented programming,** an approach that requires use of the object-message approach to the exclusion of all others.

collection (4.2.3) a grouping of one or more objects.

cousin (4.8.1) an **instance** of a different class of object that shares a common **parent object.**

daemon (3.4) a **method** invoked before or after a **primary method.** Along with **around methods,** a common approach to **method combination.**

dataflow (2.2.3) an approach to parallel processing that requires a sequential structuring of a problem's data relationships.

deep typing (4.2.4) a form of type validation that requires only that the specified **class** be among the **superobjects** of the given object.

distributed environment (5.1) an **environment** in which the **parent** and **child objects** may be scattered across two or more CPUs.

durable object (4.9) an object that remains across subsequent runs of a program. Although manipualted like other objects, the representation of a durable object is likely to resemble a database.

dynamic relocation (5.2) moving an object in a distributed simulation from one CPU to another after it is created.

dynamic typing (3.2) the **class** of an **object** is not known at compile time, but instead is only available at run time and can can be used to make decisions at that time.

environment (4.8.1) a group of objects, comprising an instance of a **parent object** and each of its corresponding **child objects.**

event (4.1) in simulation, an association of a simulated time with one or more specific actions. Also a **class** of **objects** in SIMSCRIPT family of languages.

event message (4.5.2) a Time Warp **message** that contains side-effects that change the state of an **object.**

generic object (3.5) the **superobject** in an instance-based inheritance. The term is used by both LCS and ROSS.

global virtual time (2.2.1) the minimum of the **local virtual time** of each **task**; a measure of progress in a **Time Warp** simulation.

homogeneous system (2.1) an MIMD computer which is built by combining a number of identical computers with identical roles.

implicit synchronization (2.2.1) an approach to distributed simulation that does not require the user to explicitly specify the synchronziation points for a simulation.

implied subscripting (4.2.2) a situation in which the **class** and **instance** associated with an **attribute** may be safely deduced by the compiler. Normally, only used within a corresponding **method** routine for that class.

instance (3.2) an individual copy of a **class** of **objects.**

instance-based data hiding (4.8) the use of **parent** and **child object** instances to prevent other instances of the same classes from accessing the same data.

instance-based inheritance (3.5) the **inheritance** of a property from an instance of a **generic object** by a **specific object.**

lazy cancellation (2.2.1) an implementation of **Time Warp** that attempts to minimize the use of **antimessages** during **rollback.**

local virtual time (2.2.1) in **Time Warp,** the value of simulated time, as seen by a particular **task.**

logical pointer (5.2) a mechanism for referencing an object **instance** that is independent of its hardware representation. To access the actual object, a logical pointer will normally require translation to obtain the hardware location.

message (3.3) the fundamental interaction between objects in an object-oriented program. Sent to a specific **instance** of an object by another object, it will include a message type ("selector") and may optionally send or received one or more arguments.

message forwarding (3.5) an apprproach to **instance-based inheritance** that specifies inherited behaviors on a message-by-message basis.

method (3.3) a procedure associated with a particular **message** and object **class.**

method combination (3.4) a mechanism in which a **subobject** combines its own **methods** for a **message** with those of one or more **superobjects.**

MFLOPS (2.1) millions of floating pointer operations per second.

MIMD (2.1) multiple instruction stream, multiple data stream.

MIPS (1.2) milllions of instructions per second. A measure of integer and general performance.

module (2.3) see **unit.**

monitored variable (4.1) a SIMSCRIPT II.5 variable that uses a method routine to control reads or writes to an **attribute.**

object (3.2) conceptually, the lowest level of **object-oriented programming.** From an implementation standpoint, a block of data (similar to a Pascal "record") that has associated with it a section of program. Depending on the context, this term may be used to refer to either a **class** of objects or an **instance** of one of those classes.

object handle (5.2) a form of **logical pointer** which uses a second-level indirection to refer to objects on the same CPU.

object-oriented programming (3.1) a paradigm for computation based on the specification of program states in terms of **objects** and program interactions by use of **messages.**

open implementation (3.1) referring to **object-oriented programming,** an approach that allows the use of other programming paradigms, such as those found in conventional algorithmic languages.

overloading (2.3) the declaration of a single name for two or more dissimilar uses. In Ada, **strong typing** allows unambiguous interpretation of an overloaded name.

overriding (3.4) when a **subobject** supersedes the method of its **superobject.**

package see **unit.**

parent object (4.8.1) the object that defines the scope of a **child object.**

permanent entity (4.2.2) in SIMSCRIPT, a class of objects in which all instances of the class are allocated simultaneously.

primary method (3.4) a method of a **superobject** that is completed replaced by **shadowing** or **overriding.**

private attribute (4.2.1) an **attribute** of an object that is not accessible to other objects.

process (2.3) an approach to sequencing a series of **events** associated with a single simulation object.

public attribute (4.2.1) an **attribute** of an object can be accessed by other objects, usually by sending a **message.**

query message (4.5.2) a **Time Warp** message that does not cause side-effects that change the state of an **object.**

rollback (2.2.1) the phase of a **Time Warp** simulation after a causality error is detected and before its effects have been nullified.

segment manager (5.2) a program that assigns unique identifiers to disk-based **durable objects.**

set (4.2.3) the standard ordered **collection** used by the SKMSCRIPT family of languages.

shadowing (3.4) similar to **overriding,** except used when referring to the combination of methods in a multiple-path inheritance.

shallow typing (4.2.4) a form of type checking that requires that the given object be an instance of one particular **class.**

sibling (4.8.1) another **instance** of the same class of **child object.**

specific object (3.5) the **subobject** in an instance-based inheritance. The term is used by both LCS and ROSS.

state identification template a standardized table that indicates to **Time Warp** how the state of the task must be saved.

static typing (3.2) the **class** of an **object** is fixed at compile time, an important characteristic of Pascal-family languages

strong typing (3.2) see **static typing.**

subclass (3.4) a **class** of objects that inherits properties from another class of objects, referred to as its **superclass.**

subobject (3.4) the object that inherits a property from another object. May refer to either a **subclass** or **specific object,** depending on whether a **class-based** or **instance-based inheritance** is used.

superclass (3.4) the **class** of objects containing properties which are inherited by another object class.

superobject (3.4) the object from which another object inherits a property. May refer to either a **superclass** or **generic object,** depending on whether a **class-based** or **instance-based inheritance** is used.

synchronous message (4.5.1) a message that causes the sending object to wait until the receiving object completes its corresponding **method.**

synchronous time (2.2) an approach to distributed simulation in which simulation time is maintained as a single global value for all objects and CPUs.

task (2.2.1) a single operating system job; used to refer to processes under the Time Warp operating system.

temporary entity (4.1) the basic **object** of SIMSCRIPT II.5.

throughput (2.1) the ratio of useful work done by a computer to the amount of elapsed time required to do it.

time-elapsing method (4.6.1) a **method** which may require a non-zero amount of simulated time to complete.

Time Warp (2.2.1) an approach to distributed simulation that uses **asynchronous time** and **implicit synchronization.**

type see **class.**

unit (2.3) a group of related procedures in the UCSD dialect of Pascal, and a precursor to the Modula-2 module and the Ada package.

untyped (3.2) no attempt is made to verfiy the **class** of an object, either at compile time (**static typing**) or at run time (**dynamic typing**).

utilization (2.1) the percentage of available computational power spent doing useful work.

vector processing (2.1) parallel processing achieved through expression of a problem as a matrix of related equations. Also referred to as array processing.

world map (5.2) as proposed in **Time Warp,** a translation table between a **logical pointer** and the actual CPU/machine address of an object. A copy is maintained on each CPU.

REFERENCES

[Abelson 1985] H. Abelson and Gerry J. Sussman, Structure and Interpretation of Computer Programs, M.I.T. Press, (Cambridge, MA: 1985)

[AMMO 1983] "Army Model Improvement Program Management Plan," Army Model Improvement Program Management Office (Ft. Leavenworth, KS: March 1983)

[Army 1983] "Army Model Improvement Program," Army Regulation 5-11, Department of the Army (Washington, DC: August 1983)

[ATT 1984] "Common Object File Format," UNIX System V Support Tools Guide, Release 2.0, AT&T Technologies, (Winston, NC: December 1984)

[Beckman 1984] Brian C. Beckman, "Time Warp Implementation Document," Jet Propulsion Laboratory Interoffice Memorandum 335.1-249 (Pasadena, CA: September 1984)

[Berry 1985] Orna Berry and David Jefferson, "Critical path analysis of distributed simulation," Proceedings of the Conference on Distributed Simulation 1985, Society for Computer Simulation, (La Jolla, CA: January 1985), pp. 57-60

[Birtwistle 1973] Graham Birtwistle, et al, SIMULA BEGIN, Auerbach Publishers Inc. (Philadelphia, PA: 1973)

[Birtwistle 1984a] Graham Birtwistle, et al, "Process Style Packages for Discrete Event Modeling: Data Structures and Packages in SIMULA," Transactions of the Society for Computer Simulation 1 (1), May 1984, pp. 61-82

[Birtwistle 1984b] Graham Birtwistle, et al, "Process Style Packages for Discrete Event Modeling: Using Simula's class SIMULATION," Transactions of the Society for Computer Simulation 1 (2), December 1984, pp. 175-195

[Bratley 1983] Paul Bratley, Bennett L. Fox and Linus E. Schrage, A Guide to Simulation, Springer-Verlag, (New York: 1983)

[CACI 1985] Major Military Simulations Written in SIMSCRIPT II.5, CACI, (La Jolla, CA: January 1985)

[Chandy 1981] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," Communications of the ACM 24 (4), April 1981, pp. 198-206

[Clark 1982] Randy Clark and Stephen Koehler, The UCSD Pascal Handbook, Prentice-Hall, (Englewood Cliffs, NJ: 1982)

[Concepcion 1985] Arturo I. Concepcion, "Mapping distributed simulators onto the hierarchical multi-bus microprocessor architecture," Proceedings of the Conference on Distributed Simulation 1985, Society for Computer Simulation, (La Jolla, CA: January 1985), pp. 8-13

[Cosell 1984] Bernie Cosell, et al, "An Object-Oriented Programming Facility for C," Bolt, Beranek and Newman, (Cambridge, MA: May 1984)

[Curry 1983] Gael A. Curry and Robert M. Ayers, "Experience with Traits in the Xerox Star Workstation," Proceedings of Workshop on Reusability in Programming, ITT Programming, (Stratford, CT: 1983)

[Dongarra 1984] Jack J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment," Technical Memorandum, Argonne National Laboratory (Argonne, IL: December 1984)

[Elias 1985] Antonio L. Elias and John D. Parraras, "Object-Oriented SIMSCRIPT," (Cambridge, MA: February 1985)

[Garrison 1984] William J. Garrison, NETWORK II.5 User's Manual, Version 1.1, CACI, (Los Angeles: 1984)

[Goldberg 1983] Adèle Goldberg and David Robson, Smalltalk-80: The Language and its Implementation, Addison Wesley, (Reading, MA: 1983)

[Goodhue 1985] John Goodhue, "The Butterfly Multiprocessor," Bolt, Beranek and Newman, (Cambridge, MA: January 1985)

[Gurd 1985] J.R. Gurd, C.C. Kirkham and I. Watson, "The Manchester Prototype Dataflow Computer," Communications of the ACM 28 (1), January 1985, pp. 34-52

[INMOS 1984] INMOS Limited, Occam Programming Manual, Prentice-Hall International, (London: 1984)

[Jefferson 1983] David Jefferson and Harry Sowizral, "Fast Concurrent Simulation Using the Time Warp Mechanism; Part I: Local Control," Rand Note N-1906AF, The Rand Corporation, (Santa Monica, CA: June 1983)

[Jefferson 1984] David Jefferson, et all, "Implementation of Time Warp on the Caltech Hypercube," JPL (Pasadena, CA: October 1984). Later published in Proceedings of the Conference on Distributed Simulation 1985, Society for Computer Simulation, (La Jolla, CA: January 1985), pp. 70-75

[Jefferson 1985] David Jefferson and Harry Sowizral, "Fast concurrent simulation using the time warp mechanism," Proceedings of the Conference on Distributed Simulation 1985, Society for Computer Simulation, (La Jolla, CA: January 1985), pp. 63-69

[Keohan 1984] Susan Keohan, An Introduction to Clascal, Apple Computer, Inc., (Cupertino, CA: July 1984)

[Kiviat 1968] Philip J. Kiviat, Richard Villanueva and Harry M. Markowitz, The SIMSCRIPT II Programming Language, The RAND Corporation, Report R-460-PR, (Santa Monica, CA: October 1968)

[Kozlov 1985] Alex Kozlov, "NYU's UltraComputer Network," SIAM News 18 (2), Society for Industrial and Applied Mathetmatics, March 1985.

[Koved 1984] Larry Koved, "The Object Model: A Historical Perspective," (draft) IBM Thomas J. Watson Research Center, (Yorktown Heights, NY: October 1984)

[Krasner 1983] Glenn Krasner, editor, Smalltalk-80: Bits of History, Words of Advice, Addison-Wesley, (Reading, MA: 1983)

[Law 1982] Averill M. Law and W. David Kelton, Simulation Modeling and Analysis, McGraw-Hill, (New York: 1982)

[Markowitz 1963] Harry M. Markowitz, Bernard Hausner and Herbert W. Karr, SIMSCRIPT: A Simulation Programming Language, Prentice-Hall, (Englewood Cliffs, NJ: 1963)

[Markowitz 1984] Harry M. Markowitz, Ashok Malhotra, and Donald P. Pazel, "The EAS-E Application Development System: Principles and Language Summary," Communications of the ACM 27 (8), August 1984, pp. 785-799

[May 1984] David May and Roger Shepherd, "Occam and the Transputer," Proceedings of the IFIP WG10.3 Workshop on Hardware-Supported Implementation of Concurrent Languages in Distributed Systems, North Holland Publishing Company (October 1984)

[McArthur 1982] David McArthur and Philip Klahr, The ROSS Language Manual, Rand Note N-1854AF, The Rand Corporation, (Santa Monica, CA: September 1982)

McGraw 1983] James McGraw, et al, SISAL -- Streams and Iteration in a Single-Assignment Language: Language Reference Manual, Version 1.0, Lawrence Livermore National Laboratory, (Livermore, CA: July 1983)

[Millard 1975] Wiliam Millard, "Hyperdimensional Micro-P Collection Seen Functioning as Mainframe," Digital Design 5 (11), November 1975, p. 20

[Mullarney 1982] Alasdar Mullarney, "Tabletop SIMSCRIPT Design Reassessment," CACI internal memorandum, (Los Angeles, CA: December 1982)

[Mullarney 1983] Alasdar Mullarney, editor, SIMSCRIPT II.5 Programming Language, CACI, (Los Angeles: 1983)

[Mullarney 1984] Alasdar Mullarney, et al, PC SIMSCRIPT II.5 User's Manual, Release 1.2, CACI, (Los Angeles: December 1984)

[Nugent 1983] Richard O. Nugent, "A Preliminary Evaluation of Object-Oriented Programming for Ground Combat Modelling," Working Paper 83W00407, The MITRE Corporation, (McLean, VA: September 1983)

[Russell 1969] Edward C. Russell, "Automatic Program Analysis," UCLA School of Engineering and Applied Science, Report 69-12, (Los Angeles: March 1969)

[Russell 1983] Edward C. Russell, Building Simulation Models in SIMSCRIPT II.5, CACI, (Los Angeles: 1983)

[Seitz 1985] Charles L. Seitz, "The Cosmic Cube," Communications of the ACM 28 (1), January 1985, pp. 22-33

[Stallman 1984] Richard Stallman, Daniel Weinreb, and David Moon, Lisp Machine Manual, Sixth Edition, MIT Artificial Intelligence Laboratory, (Cambridge, MA: June 1984)

[Stroustrup 1984a] Bjarne Stroustrup, "Data Abstraction in C++," Bell Labs Technical Journal part 2, October 1984

[Stroustrup 1984b] Bjarne Stroustrup, The C++ Programming Language -- Reference Manual, C++ Release E Documentation, AT&T Bell Laboratories, (Murray Hill, NJ: November 1984)

[Tesler 1985a] Larry Tesler, "Object Pascal Report," Apple Computer, Inc., (Cupertino, CA: February 1985)

[Tesler 1985b] Larry Tesler, "Object Pascal vs. Lisa Clascal," Apple Computer, Inc., (Cupertino, CA: February 1985)

[West 1984] Joel West and Glen Johnson, SIMSCRIPT II.5 User's Manual for VAX/VMS, Release 4.3, CACI, (Los Angeles: May 1984)

[West 1985] Joel West and Timothy Lynch, UNIX SIMSCRIPT II.5 User Manual, Release 1.2, CACI, (La Jolla, CA: July 1985)

[Wilson 1984] Pete Wilson, "Highly Concurrent Systems Using the Transputer," INMOS Technical Note 5, INMOS Corporation, (Colorado Springs, CO: 1984)

[Ziegler 1985] Bernard P. Ziegler, "Discrete event formalism for model based distributed simulation," Proceedings of the Conference on Distributed Simulation 1985, Society for Computer Simulation, (La Jolla, CA: January 1985), pp. 3-7

APPENDIX A: A SUMMARY OF LCS CONCEPTS

The following sample statements use descriptive names to summarize the fundamental object-oriented concepts of a Language for Concurrent Simulation. LCS keywords are shown in lower case, and the language keyword for the corresponding property is underlined. User-defined values are shown in upper case, and the key value is shown in bold. The use of elipses indicates an incomplete statement or program fragment.

A.1 Objects (classes)

every **OBJECT_CLASS** ...

A.2 Public Attributes

every OBJECT has **ATTRIBUTE**

A.3 Collections (sets)

every OWNING_OBJECT owns **COLLECTION**
every MEMBER_OBJECT belongs to **COLLECTION**

A.4 Message passing

tell OBJECT **MESSAGE**(ARGUMENTS)
ask OBJECT **MESSAGE**(ARGUMENTS)
let ... = OBJECT:**MESSAGE**(ARGUMENTS)

A.5 Message receiving

method **MESSAGE_TYPE** for OBJECT_CLASS
...
end

A.6 Class-based inheritance

every SUB_OBJECT is **SUPER_OBJECT**

A.7 Instance-based inheritance

```
every GENERIC_OBJECT refers to SPECIFIC_OBJECT
```

A.8 Private Attributes

```
object PARENT_OBJECT
     define PRIVATE_ATTRIBUTE as ...
     ...
end
```

A.9 Parent and child objects (Object Environments)

```
object PARENT_OBJECT
     every CHILD_OBJECT ...
     ...
end
```

APPENDIX B: EXISTING OBJECT-ORIENTED LANGUAGES

This chapter is a survey of some significant object-oriented languages and object-oriented extensions to existing languages. The Appendix describes two object-oriented languages, SIMULA and Smalltalk-80, object-oriented extensions to C and Pascal, and two packages written in Lisp to enable writing object-oriented simulations: ROSS and the Flavor System.

Several languages have been omitted, perhaps arbitrarily. C/Flavors, described in **[Cosell 1984]**, is an implementation of the Lisp Flavor System (Section B.6) in the C language. It is less C-like than C++ (Section B.3), although both languages retain existing concepts for ordinary algorithmic functions.

Docmentation for the Traits package of the Mesa programming language is less readily available than for the extensions to other structured languages described in this chapter. However, the system does provide true multiple inheritance in a compiled language, one of the primary goals of a Language for Concurrent Simulation. The experience of **[Curry 1983]** is instructive, particularly in the area of a programming environment to support compile-time resolution of state variable locations.

B.1 SMALLTALK-80

The prime example of a closed object-oriented programming language, Smalltalk-80 is the successor to the early prototypes of active object implementations, such as Actor. A complete definition of the language can be found in **[Goldberg 1983]**.

Smalltalk is more of a system than a language, including the editor, graphics manager, compiler, interpreter and the operating system itself in an integrated package. The single most striking feature of Smalltalk is that, being a closed implementation, all entities in the language are objects (including numbers) and all operations are performed by message-passing. The first consequence is that there is no special construct for sending a message; the basic Smalltalk sentence is of the form:

receivingObject messageName argument1 argument2...

In Smalltalk nomenclature, **receivingObject** is sent the message **messageName argument1 argument2**, and **messageName** is called the message selector (we will call it the message name). Sometimes, the result may look extremely algebraic; for example:

x <- 3 + 4

results in the assignment of the object **7** as the value of the object **x**; however, this is obtained by sending the object **3** the message **+4**, where **+** is the message name and **4** is the argument. Objects of class number understand all the algebraic operation messages of course, but the analogy breaks down when performing trigonometric operations:

x <- theta sin

In addition to these simple message formats, Smalltalk allows message names to be "split" between the arguments. For example, instead of having a form such as:

personalAccount addNewExpense 34.65 rent check

where **34.65** and the objects **rent** and **check** are the arguments of the **addNewExpense** message sent to **personalAccount.** Instead, Smalltalk allows you to have a compound message name made up of **spend: for:** and **by:** in the following way:

personalAccount spend: 34.65 for: rent by: check

where all three keywords **spend:**, **for:** and **by:** make up the message name, so that **spend: for:** and **when** would be interpreted as a different, unrelated message. This is a readability feature.

As documented by **[Goldberg 1983]**, the first version of Smalltalk-80 allows objects to be structured in a simple-tree hierarchy only; that is, each object has one and only one superobject above it, from which it inherits both attributes and methods. However, Version 2.0 provides for multiple inheritance, although the authors feel that current syntax does not adequately deal with the added conceptual complexity.[1]

Shadowing, (called <u>overriding</u> in Smalltalk) is possible for both attributes and methods, and is performed by simply defining a similarly-named attribute or method at the lower level class.

There is also message deferral. In a method, an object of a given class may send to itself a message it cannot handle, but that one of its subclasses will. For example, a method for the class **figure** may send itself the message **rotate** even if there is not **rotate** method defined for **figure.** The object "knows" there will never be actual instances of simply **figure**, but only of subclasses (such as **triangle** and **square**) which include the **figure** class, as well as methods for **rotate.**

There is the inevitable object **self** (called a pseudo-variable in Smalltalk) which can be used by a method to refer to the current instance. There is also the very interesting object **super**, which is like **self**, but which, when sent a message, starts looking for the methods starting with the object's first superobject, thus allowing the programmer to override method shadowing in the code. Actually, this feature is not a frill, but is required in Smalltalk to properly interact with the message deferral mechanism.

The characteristics of a class of objects, that is, their attributes and methods, are determined by a "template" that is itself an object of class -- what else -- **Class.** Instances of objects of a given class are created by sending this object the **New** message. Inevitably, there are classes of **Class** objects, leading to the concept of "Metaclasses," all the way up to the original root class, **Object.** This is a result of the closed object-orientation of Smalltalk and should be of no consequence in the design of an open implementation such as LCS.

Instance variables are truly local; only the methods for a particular object class can access that object's instance variables (and then only for the particular instance that received the message). All cross-accesses, if required, must be performed by message-passing. Messages are bi-directional, that is, they may return a value. Since everything in the Smalltalk universe is an object, there is no need for an "object mode" variable: integers, reals and strings are "special cases" of objects.

In addition to the instance variables (attributes) of objects, Smalltalk allows the following scoping of variables:

1. Temporary variables, dynamic variables within methods similar to SIMSCRIPT event procedures' local variables.

2. Class variables, a unique concept of variables shared by all the instances of a single class. Class variables can be conceptualized as "belonging" to the appropriate class object. Although they can be accessed by a method as if they were an instance variable, they are initialized by having the message **init** being sent to the class object. Therefore, the user may want to create an **init** method for the class object to perform this initialization, as if they were instance variables of the class object. This is all very simple, elegant and confusing.

3. Global variables, accessible to all objects; since all objects are, in the last resort, members of a super-superclass **Object**, globals can be conceputalized as class variables of this superclass.

4. Pool variables, a scoping of variables somewhat between class and global. They are shared by all instances of a set of classes.

The combination of restricting the object hierarchy to a simple tree and the existence of class variables allows for a very elegant combination of class inheritance and instance inheritance. Suppose that we have a class **soldier**, whose instances may belong to a blue army or a red army. All blue soldiers will have "m16" as the value of the instance variable **pointWeapon**, and all red soldiers will have the value "ak47" for that attribute.

Instead of having a true instance variable **pointWeapon**, it is possible to create two "Dummy Classes", **blueSoldier** and **redSoldier**, with no instance variables (and no methods), but with the class variable **pointWeapon** initialized to the two different values. Any method for the superclass soldier may, by deferral (see above), make reference to the variable **pointWeapon**, and the appropriate value will be picked up if the instance actually handling this message belongs to the blueSoldier or redSoldier "dummy" subclasses.

Finally, it is very important to note that Smalltalk-80 includes, in the basic package, a large number of artifacts designed to support discrete-event simulation; there is a class **SimulationObject**, which includes messages such as (the titles are suggestive):

1. **startUp**
2. **tasks**
3. **finishUp**
4. **holdFor**
5. **scheduleArrivalOf: at:**
6. **scheduleArrivalOf: after:**
7. **resume**
8. **acquire**
9. **release**
10. **produce**

The last four messages relate to **resource** class objects, of which there are two variants: **static**, which corresponds more or less to the SIMSCRIPT II.5 resources, and **coordinated**, which allows the resource object itself to have intelligence with which to implement, for example, preemption.

B.2 SIMULA

Unlike Smalltalk, SIMULA is not a closed implementation; rather, it is an extension of the Algol-60 language and subsumes Algol-60 with a few intended exceptions. SIMULA is a rich compiled-only language; it has been implemented on a number of systems **[Birtwistle 1984a]**. Indubitably, it would have received more attention and usage in the U.S. were it not for its extremely poor documentation, lack of readable textbooks and a worldview best summed up as "inordinately complex" **[Bratley 1983]**. The following analysis is based on a study of reference **[Birtwistle 1973]**.

From Algol, SIMULA inherits its dynamic calling mechanism, which enables the writing of recursive procedures, as well as its powerful structure and character manipulation primitives, which are enhanced in SIMULA as well. Object-oriented programming is achieved by means of an object-mode variable, similar to the kind proposed for LCS. Nominally, SIMULA object variables are explicitly typed, as in the following declarations (SIMULA reserved words are typically underlined by the SIMULA compiler in the compilaton listings):

REF(POINT)P1,P2,P3

indicating that **P1**, **P2** and **P3** are object variables that can only reference an object of class **POINT**. Since all object mode variables must be thus qualified, it would seem that all the type-sensitive questions could be resolved at compile-time and that there is no need for run-time knowledge of the type of current value object of an object variable.

However, SIMULA allows an object variable to contain an object of a superclass of the class that is nominally declared for that variable. For example, if **CENTRE** is a superclass of **POINT**, then **P1** can have a **CENTRE**-class object as its value. There are also facilities to determine the object type of the value of a variable, both shallow (IS) and deep (IN), and a case-dispatch construct for dispatching on object type (INSPECT). Clearly, this requires run-time knowledge of the object type of the value of object variables.

Attribute and behavior inheritance is strictly tree-structured. Any class has a single parent class which is stated in the class declaration by means of a prefix:

```
LOCATION CLASS AIRCRAFT(CARRIER,FLIGHTNO);INTEGER FLIGHTNO;
          REF(AIRPLANE) CARRIER;
BEGIN REF(LOCATIO) ORIGIN,DESTINATION;
          REAL FLIGHTTIME;
                    PROCEDURE FLY........
            END***FLY***;
           PROCEDURE LAND.......
            END***LAND***
ORIGIN:-CARIER.MAINBASE;
DESTINATION;-CARRIER.DESTINATION(FLIGHTNO);
FLIGHTTIME:=ORIGIN.GCDISTANCE(DESTINATION)/
FLIGHTSPEED;
END***AIRCRAFT***;
```

In the previous example, **AIRCRAFT** is a subclass of **LOCATION**, and thus it inherits **LOCATION**'s attributes and methods. Note that some of **AIRCRAFT**'s own attributes can appear as "parameters" of the class name. This allows these attributes to be initialized by the statement that constructs the object. For instance, a new aircraft will be created by:

```
NEWAC :- NEW AIRCRAFT(TW,611);
```

Note also that the methods are specified as procedures within the lexical scoping of the block declaring the class and that the "maincode" (corresponding to the SIMSCRIPT process procedure) is specified as the "body" of the class block.

The above example also illustrates that period (".") within object attributes has a different meaning than in SIMSCRIPT. In the former case, it serves as an abitrary identifier character, while in SIMULA, the period represents (as in Pascal and C) a binary dereferencing operator. A composite name separated by a period, such as **CARRIER.MAINBASE**, represents accessing the **MAINBASE** attribute of the **CARRIER** object.

Of more interest is the way in which methods are invoked -- that is, the way messages are passed. They look like function calls qualified by the object variable that will receive the message and thus look very similar to attribute access. The **ORIGIN.CGDISTANCE(DESTINATION)** statement sends the **CGDISTANCE** method to the object, which is the value of **ORIGIN**, with the value of **DESTINATION** as a parameter.

The SIMULA documentation does not use the Actor paradigm to explain message passing; rather, they favor the analogy to function calling, which is emphasized by the block structure of the class and method definition syntax. It is possible to chain messages at the source code level if the value they return is an object such as

```
VALUE := OBJ1.MESSAGE1.MESSAGE2(PARAMETER);
```

which sends **MESSAGE2** to the object returned by **MESSAGE1** when sent to OBJ1.

Note also the use of the symbol **:-** to assign object values to object variables, in contrast to the Algol **:=** symbol used for numerical and text variables; similarly, SIMULA uses == and =/= for the boolean tests for object equality and inequality, respectively, while = and ^= are used for all other variable modes. This is done to improve readability, at the expense of increasing the likelyhood of typographical errors.

Shadowing and deferral are allowed in SIMULA, but they must be explicitly authorized by the superobject for each attribute and method to be shadowed. To authorize shadowing, the attribute, or method, must include the keyword VIRTUAL in that object, even if it has a null body.

It is possible to have a simple form of before- and after-method, limited to the maincode (i.e. the **init** method). When an object is instantiated, the maincode of the highest (most primitive) component class is executed first, then the next one down, and so on until the current class is reached. However, if the keyword INNER is included as a "statement" in a class's maincode, its execution is interrupted, and the successor class's maincodes are executed before going on with the next statement. A method may use the pseudo-variable **THIS** to refer to the current instantiation of the object, while **NONE** is used as the null object.

B.3 C++

A very recent development has been C++, which is similar to SIMULA in two important ways. First, like SIMULA, the object-oriented features are an extension to an existing algorithmic language. Second, as its author acknowledges, those features are largely derived from those of SIMULA67.

With a few minor exceptions, C++ is a superset of the C language, an extremely compact language designed for portable implementation of systems software, including the UNIX operating system. C++ is implemented as a pre-processor to the standard C compiler included with AT&T's UNIX systems. The language has been developed at AT&T Bell Laboratories by Bjarne Stroustrup since the early 1980s. Various incarnations have been distributed within AT&T since then. More recently, the package has been made available to universities.

According to its author,

> C++ classes distinguish themselves by combining facilities for creating class hierarchies with efficient implementation. The facilities for object creation and initialization are notable. The facilities for overloading assignment and argument passing are unique for C++ **[Stroustrup 1984a].**

Stroustrup contrasts C++ classes to Smalltalk by noting "while a C++ base class provides a fixed type-checked interface to a set of derived classes, a Smalltalk superclass provides a minimal untyped set of facilities that can be arbitrarily modified ... all functions are virtual and all type checking done at run time."

As proposed for LCS, C++ takes two approaches for involving object manipulation methods, in which the corresponding method is selected at compile or run-time.

If **func** is invoked for **object** with parameter **parm**, this is expressed in C++ in a manner similar to SIMULA67, e.g.

object.func(parm)

If the function is strongly typed, only one possible class of object can be passed. This translates to the standard C notation

function(&object,parm)

where the ampersand represents C's "take the address of" operator.

Behavior inheritance is implemented through <u>virtual</u> functions. For example, the function **move** could be defined for the class **vehicle.** The declaration

```
extern static location home; /*global and unscoped*/
class moving_object
{       location    position
        float       maxspeed;
        float       speed;
        float       heading;

public:
        void        floor_it{}
            {speed = maxspeed;}
        virtual     void move(location);
        void        go_home();
        ---
};
```

defines a class of objects (**moving_object**) for which the function **move** is defined to exist, but it is a virtual function, for which the definition of the function behavior is deferred to a subclass. A new subclass, **vehicle**, can be built upon the framework of a **moving_object** as follows:

```
class vehicle:public moving_object
{   float   gross_veh_weight;
        ...
    public:
        void    move(location);
        ...
        };
```

A sequence

```
vehicle car;
        ...
    car.floor_it();
```

would cause the **floor_it** subroutine for the class **moving_object** to be called for **car**, since a separate behavior was not defined for the **vehicle** derived type; this routine is referred to as **moving_object::floor_it()**.

However, the sequence

```
location destination
    ...
car.move(destination);
```

would cause the routine **vehicle::move()** to be invoked for the object **car**.

A different set of declarations could be used to define a **move** function for another subclass of **moving_object**, such as **airplane** or **person.** Then, the **moving_object** function could be defined

```
void go_home()
   {  this.move(home);
    }
```

where **this** is the same as in SIMULA, and similar to the **self** of Smalltalk or Flavors. The following sequence of code would do exactly as expected:

```
vehicle     car;
airplane    jet;
person      boss;
          ...
if (time==done_time)
  { car.go_home();
    jet.go_home();
    boss.go_home();
  }
```

Even though the function **go_home** is defined for all subclasses of **moving_object**, it must dispatch the **move** behavior at run-time based on the object subclass for **this.** This dispatching is done via a table look-up of a subroutine address during execution of the statement **this.move(home).**

As with SIMULA67, the hierarchy of classes and inheritance is strictly tree-structured. A C++ class must be derived from a single class; the multiple inheritance of Flavors is not allowed.

B.4 Object Pascal

Object Pascal represents a recent collaboration between Niklaus Wirth, the original author of Pascal, and the team at Apple Computer that developed Clascal for the latter's Lisa microcomputer. The authors of report **[Tesler 1985a]** explicitly place the specifications of the new language in the public domain, and encourage others to develop compilers implementing those specifications.

The Clascal language, described in **[Keohan 1984]**, has been taught by Apple instructors for the past two years. Object Pascal is intended to correct the problems found in using and teaching the earlier language. In fact, one of the primary goals of the Object Pascal has been ease of learning, and this was a factor in the deliberate decision to exclude multiple inheritance.[2]

Despite the independent research efforts involved, Object Pascal is in many ways reminiscent of C++. Both are fully-compiled languages, extend existing structured languages are based on the class concepts of SIMULA.

Both mimick SIMULA's **object.method(arguments)** syntax for message passing, use the **virtual** keyword for deferral of method implementation to a subclass, and allow omission of **self** in a method routine, with the assumption that methods refer to **self** unless otherwise noted. As noted in **[Tesler 1985b]**, this change from Clascal makes it easier to convert conventional Pascal code written with global variables and procedures.

According to Apple's announced plans, the major use for Object Pascal will be in supporting libraries of inherited behaviors for developing Macintosh software. Apple can supply a set of standard behaviors to software developers, and then those behaviors can be selectively included, enhanced, or replaced as appropriate for the particular application. This "MacApp" system is also planned to include development in C and other programming languages.

B.5 Ross

ROSS is an extension of the Lisp language developed at the Rand Corporation, specifically to allow easy object-oriented coding for discrete-event simulations. From Lisp, ROSS inherits the immense power of symbolic manipulaton which makes, for instance, special object-mode variables -- necessary in SIMULA and in LCS -- totally superfluous. It is also capable of inheriting the powerful interactive software environments of some Lisp implementations, although this depends strongly on the particular Lisp system being used.

Sadly, it also inherits the basic problem of Lisp: currently available Von-Neumann type machines are woefully inefficient in performing the fundamental Lisp operations (for which truly associative memory hardware would be required, a problem it also shares with Smalltalk). Although specialized processors have been developed to aid in the emulation of this hardware ("Lisp Machines"), the same amount of money spent on conventional hardware will execute conventional software at a faster rate (approximately four times as fast, according to **[Elias 1985]**). Clearly, Lisp and Smalltalk are desirable when lower software development costs are more important than execution speed.

The following analysis of ROSS is based on references **[McArthur 1982]**.

ROSS appears to the programmer as a very simple extension of Lisp; indeed, there is only one new verb, three predefined objects, and 10 "properties" (properties are a generic Lisp facility) associated with ROSS. ROSS objects appear to be simple Lisp objects; messages are sent by means of the "tell" Lisp verb (which has the alias "ask"). Method definition, class definition, and instance creation all are achieved by means of messages, as opposed to language "primitives," as in SIMULA or Flavors. A typical message passing would look like:

(tell object1 body-of-message)

In ROSS, there are no object classes; instead, there is a hierarchy of related objects, where objects inherit the attributes and behavior from the objects above them in the hierarchy by forwarding the messages they cannot handle. Since there are no explicit classes, there is no concept of "instance" as in the other languages described in this chapter.

A distinction is made between objects that have other objects as sub-objects, and instances that have no sub-objects; the former are called generic objects in ROSS, and the latter instance objects. Apart from their location in the tree (nodes or leaves), there is no qualitative distinction between these two kinds of objects. Indeed, an object may be created without either the instance or generic specification (by means of the **MAKE** message) and given the appropriate status later.

New object creation is performed by sending an appropriate message (for example, **CREATE**) to another object, as in Smalltalk. This necessitates the preexistence of a fundamental object, called **SOMETHING** in ROSS (the other three predefined objects are the simulation clock and two debugging aids).

The attributes of newly created objects are determined by parameters of the **CREATE** message. Thus, a "class" of objects having the same attribute structure is possible by having the code CREATE all these objects in the same way. Method sharing, however, is achieved by means of common parent objects.

Thus, two different mechanisms are used to inherit the state and functional components of behavior. A true class in the SIMULA and Smalltalk sense can be created by a combination of these means, but must be explicitly performed in the code, and there is no explicit description of such a class in the resulting ROSS code.

The most advanced characteristic of ROSS, and the one that sets it apart from the three other languages analyzed here, is its pattern-matching based message dispatching scheme. Rather than the fixed message names of Smalltalk, SIMULA and the Flavor System, ROSS triggers methods by means of complex patterns including both fixed and "variable" elements. This can only be illustrated by defining a method, which is accomplished by sending a message to an (presumably GENERIC) object which in question, and includes the keyword **WHEN:**

(tell object1 when receiving (message-pattern)
....body of method....)

The message pattern consists of "fixed" items (items that must match the incoming message literally for a match to be successful), and "variable" items (items that will match anything, but that will remember what they matched for the duration of the method). For example, the pattern:

(press the >thing)

will match any message whose first two tokens are "press" and "the," while the third token can be anything at all; however, the variable "thing" will be bound, in the body of the method, to this third token (itself a Lisp object, therefore, a potential ROSS object). For example, if an object is given the following method definition:

(tell generic-object when receiving (press the >thing)
(print thing)

Then if that object -- or one of its descendents -- is sent the message:

(tell object-instance press the button)

then **button** will be printed, since **thing** will be bound to **button** in the body of the method when it executes (note that **print** is a Lisp primitive; the body of the method may contain any Lisp statement including, of course, **ask**).

This pattern-matching mechanism is exceedingly powerful; it allows messages to be constructed in a way very reminiscent of natural English (as opposed to the "Englishese" of COBOL and SIMSCRIPT II.5); at the same time, it imposes an exceedingly heavy run-time computational burden which cannot be alleviated by smart compilaton (indeed, powerful pattern-matching approaches demand specialized, dedicated hardware).

Once the mechanism for dispatching messages has been explained, let us return to the instance-based inheritance mechanism of ROSS. ROSS implements attributes dynamically: a new attribute can be added to an existing object instance and an existing attribute can be expurgated from an existing instance. This can be interpreted either as a very powerful capability or a source of programming problems, since the structure of a given object can change during runtime. Clearly this requires the dynamic nature of Lisp system, and cannot be implemented by "one-time" compilation approaches, such as SIMULA or SIMSCRIPT.

The effective attributes of a ROSS object are determined in two ways: the individual object's variables, (which would correspond to instance variables in a class-based system), are specified in the message that is sent to create the object; but the object also "inherits" the instance variables of its parent objects (ROSS allows multiple parents, like Flavors, and unlike SIMULA, C++ and Object Pascal.)

When a method references a variable, it is first looked up among the instance's own instance variables (as defined when it was created); if not found, the pare(s) instances are searched. In this case, the variable acts like a Smalltalk "Class variable," in that is a value shared by all the members of that **class** (actually, all the offspring of the instance that actually contains the value). Implicit in this scheme is the capability of shadowing; attribute merging, in the Flavors sense, is, however, impossible. The same mechanism applies to methods.

ROSS's lack of an explicit class mechanism is a severe obstacle to its practical application as a production simulation language, second only to the run-time overhead imposed by Lisp and the pattern-matching message dispatch system.

B.6 The Flavor System

The so-called "Flavor System" is a package written in Lisp to implement object-oriented programming. Developed originally for the "Lisp Machine" Lisp **[Stallman 1984]**, it was subsequently ported to the NIL VAX/VMS implementation of Lisp **[Burke 1984]**. It has been used extensively by the authors of **[Elias 1985]** for air traffic control simulation; this section is based on that experience.

While an obviously open system, it is interesting to note that the NIL language itself system was developed entirely in object-oriented style. An ingenious "bootstrapping" scheme was used, whereby the "core" of NIL was simulated in Maclisp on a PDP-20 and made to produce VAX-11 code. It was then transferred in binary form to a VAX. With this core, the rest of the NIL system, including the editor, interpreter, and user-level constructs, was developed.

Thus, while the Flavor system is truly an "add-on" in Lisp Machine Lisp, it is an intrinsic component and a basic development tool of the NIL language system. For these reasons, we will describe mainly the NIL implementation of Flavors, rather than the Lisp Machine one.

Principal features of the Flavor system are:

1. Class-based inheritance is provided -- unlike the Lisp-based ROSS, but similar to the other object-oriented languages discussed previously.

2. Flavors supports multiple-path inheritance. The class structure can be arbitrarily specified and is not limited to a simple tree structure, as is the case in SIMULA, C++ and Object Pascal. The user is allowed even to specify apparently cyclic class definitions (such as "class foo includes class bar which, in turn, includes class foo"). The system actually unravels and "cuts" the cycles at execution time (it is difficult to differentiate between "compile time" and "run time" in NIL, since it is capable of true incremental compilation and linking).

3. A large repertoire of attribute and method combination alternatives are included: for better or for worse, this is the consequence of the multiple inheritance capability. The system is both very powerful and very complex: only very sophisticated users can take full advantage of its performance.

4. The Flavor system uses syntactically separate definition of object classes and methods. While both Smalltalk and SIMULA syntactically group the definitions of object classes and the method functions, the Flavor System (as with Object Pascal) treats class and method definitions as separate operations, with the obvious restriction that a class must have been defined before a method for that class can be defined. This is required by the interactive, incremental programming nature of Lisp; a user must be able to add a new method to an existing class of objects while running the code.

The Flavors design is capable of reasonable runtime performance on conventional architectures when coupled with an exceptionally efficient Lisp implementation, such as NIL. For example, object-oriented programming is used to implement a variant of the Emacs editor in NIL, and the editor's performance is acceptably fast, even in a multi-user VAX configuration.

The Flavor System is implemented in NIL using the following four constructs:

defflavor This verb is given as arguments a list of attribute name. All variables in NIL are dynamically typed, so there is no need to declare the type of the attributes. Arguments also include a list of component flavors (the superclasses) and various options controlling the automatic creation of utility methods, such as attribute accessors and mutators.

defmethod This verb defines a method for a given message and class, and is other wise identical to a normal Lisp function definition construct. The pseudovariable **self** is used to reference the currently active object, and there is no equivalent to Smalltalks **super** pseudovariable.

send This verb causes a message to be sent to the object which is its first argument; the next argument is the message selector, and the remaining arguments the parameters that the selected method will receive. As is customary with Lisp, methods return a function value.

make-instance This verb returns a newly-created instance of the class specified as the first argument. Parameter-type arguments may be used to specify initial values for the attributes of the new instance.

One of the most striking features of the Flavor system is the tremendous flexibility built into it. The options available to the user in the definition of a flavor allow him to control in great detail the internal operations of the system itself. Examples of this are a vast set of options for combining methods and ways to override just about every aspect of system-defined processing. Useful as it may be to the sophisticated user, this power may, however, overwhelm less experienced or skilled users.

Appendix Notes

1. Personal communication, Adele Goldberg of Xerox Palo Alto Research Center (PARC), April 8, 1985.

2. Personal communication with Larry Tesler, Apple Computer Inc., March 23, 1985.

CACI

3344 North Torrey Pines Court, La Jolla, California 92037 (619) 457-9681