1990

Distributed object-oriented discrete event simulation

Barbara Hendry

Rochester Institute of Technology
School of Computer Science and Technology

Distributed Object-oriented Discrete Event Simulation

By Barbara Hendry

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by:

_____
Robert T. Gayvert

_____
John A. Biles

_____
Harvey E. Rhody

_____
Peter G. Anderson

August 15, 1990

SAMPLE statements to **authorize** or **deny** permission to reproduce on RIT thesis

1. Title of thesis Distributed, Object-oriented Discrete Event Simulation

I _____ hereby **grant permission** to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date 10/8/90

2. Title of thesis _____

_____

I _____ **prefer to be contacted** each time a request for reproduction is made. I can be reached at the following address.

_____

_____

Date _____

3. Title of Thesis _____

_____

I _____ hereby **deny** permission to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part.

Date _____

rev. 1/89

Abstract: This paper presents criteria for an 'ideal' simulation language, compares four traditional simulation languages to this ideal and concludes that an object-oriented approach to simulation comes closer to the ideal than the traditional procedural approach. It also examines how the object-oriented approach can be very beneficial for distributing a simulation problem among several machines. A distributed object-oriented package is described and a manufacturing example written and explained using this package.

Key Words: Discrete event simulation; Object-oriented languages; Distributed systems; Time Warp

Acknowledgements: I would like to thank my thesis advisor, Al Biles, and the rest of my thesis committee, Harvey Rhody and Peter Anderson, and especially, Rob Gayvert, for their invaluable help.

TABLE OF CONTENTS

Chapter 1 - Introduction

The purpose of this thesis is to explore the object-oriented approach to distributed discrete event simulation.

Initially, criteria are discussed as to what would make an ideal simulation language: what functionality is necessary to reap the largest possible benefits from simulation.

Then, four traditional simulation languages, GPSS, Simscript, SLAM and Simula, are analyzed and evaluated in regard to the criteria already developed. To illustrate further the major features of these languages, a classic factory simulation problem is solved and examined in GPSS, Simscript and SLAM. Based on this examination and analysis, these languages do not fulfill many of the requirements described earlier for an ideal simulation language.

Searching for a strategy with more of the desired functionality, the object-oriented approach is explored. The key concepts are outlined and several languages in common use, Smalltalk, Flavors, C++, and Ross, are described and evaluated. Compared with the ideal described initially, this approach seems to achieve many more of the goals of discrete event simulation than did the traditional strategies.

An increasingly important criterion, as very large and complex systems are simulated, is the capability of a programming approach to distribute a problem over several machines. The topic of distributed simulation is examined with emphasis upon the problems of synchronization among machines. Two main approaches are described: the

conservative Chandy-Misra strategy and the more optimistic Time Warp with rollback.

To illustrate the advantages of the object-oriented approach for distributed simulation, a distributed discrete event simulation package was designed and implemented based on an existing simulation package written in Flavors on the Texas Instrument Explorers at RIT Research Corporation. Using this distributed package, the manufacturing problem from above was implemented to parallel the sample programs outlined before in GPSS, Simscript and SLAM. Use of this distributed package is then evaluated based on the criteria established previously, with special emphasis upon the ability of the simulation to be distributed, efficiency, use of graphics, ease of understanding and design, capacity for intelligent exploration and ease of modification.

Conclusions are drawn asserting the merits of the object-oriented approach to simulation, with special emphasis upon its ability to be distributed.

Chapter 2 - Background Information on Simulation

Simulation, modeling a dynamic system and observing its behavior over time (Li, 1987), is a powerful tool. It can aid its users in better understanding and predicting a system's structure and behavior, is useful when it is impossible or too time consuming, costly, dangerous or awkward to use the real system, and allows repeated experimentation with a system under controlled conditions. Unfortunately, all the potential advantages of simulation are not being tapped with traditional simulation methods.

Let us explore traditional simulation approaches, desirable features in simulation languages and what methods could be employed to make simulation more useful. Simulation is commonly divided into discrete simulation and continuous simulation. In discrete simulation, variables representing the state of the simulation only change at discrete points in time, while in continuous simulation, the state variables may change continually. We will concentrate on discrete simulation.

2.1 Necessary Features of Simulation Languages

In order to reap all the benefits of simulation, certain features must be present in a simulation language.

A simulation language must be able to represent the real world components of a system comprehensibly and concisely with explicit, examinable and modifiable assumptions. The language must employ user-friendly, consistent, unambiguous and English-like syntax, and clear and well-defined semantics. Fulfilling these two requirements will make it simpler

and faster to construct and debug an effective simulation. Because it will be easier to judge whether the simulation reflects the underlying system, especially for non-programming experts, fewer mistakes are likely to be made during the development, fine-tuning and maintenance of the program. The language also should include an easy-to-use mechanism for defining and manipulating powerful data structures, an efficient but easily changed timing mechanism, and a battery of good random number generators.

Such an ideal simulation language would have to be reasonably efficient for complex models. If the likely scale of such models is considered, the need for running a simulation concurrently becomes clear. For example, simulating an electronic telephone switching system, where a switch generates about one hundred internal messages when completing a local call, might require, for 15 minutes of simulated time, nearly ten million messages (Misra, 1986). Currently, on modern machinery, this might take several hours. Concurrency becomes an obvious need but such concurrency should be transparent to a user.

Also required would be good facilities for selectively collecting statistics and printing comprehensible results. Users should be able to assert easily what kinds of statistics are needed and in what form they should be provided. Commonly needed statistics should be provided simply, but detailed and correct production of more complex and less frequently required data also should be permitted. In order for these results to be easily and conveniently stored and analyzed, a database management system would need to

integrate properly with the program.

Full interactive debugging facilities, rather than just the capacity to trace certain values, would allow a user to more quickly produce a working simulation, thus increasing the user's productivity. Such facilities should allow a user complete control over a simulation's execution and access to data.

A key requirement for such a language would be a capacity for easy modification. A program should be constructed in a modular fashion so that a user can change one part or add an additional functional component without altering the rest of the system. A user should be able to construct programs out of existing parts and create alternative models by adding and deleting sections incrementally. Ease of modification would be furthered by the creation of pre-compiled macros to make a model more concise, easier to debug, and faster to compile.

In order to make simulation a more viable and efficient tool, a simulation language should also have facilities for intelligent exploration so that the user can control the level of aggregation, stop an execution, change parameters, or back up and try an alternative route. This is vital in order to fully explore the system being examined.

Graphics and animation can serve to illustrate more clearly what is happening while a simulation is executing. The flow of processes occurring along with basic statistics about the system could be illustrated and aid in understanding what is happening. This could help in debugging and in communicating results to viewers.

Such a list of requirements for a good simulation

language is obviously an ideal. Certain of these requirements might need to be balanced against each other, for example, readability vs. efficiency, graphical views during simulation vs. concurrency. However, in order to decide what is the best way to reach the potential for simulation, simulation languages need to be carefully evaluated in regard to these requirements. As new approaches to simulation have been developed, new possibilities for simulation have emerged; the user should be able to expect more from a simulation and a simulation language.

2.3 Discrete Event Simulation Languages

Discrete event simulation languages have generally taken one of three world views to model a system. Event-oriented languages focus on the occurrence of events and the resulting changes in the state of the model. Activity-oriented models emphasize the activities that a system performs while process-oriented languages stress the processes through which entities in the system flow. Event scheduling provides greater power and flexibility than the process-oriented approach but is more tedious to write. Process interaction can be more precise and easier to construct but is limited to the standard blocks or nodes provided, unless subroutines written in other languages are used to model less common structures (Haider 1986).

There are many, many languages and packages used to do simulation. However, three languages, GPSS, Simscript, and SLAM, a Fortran-based language, are the most common and are general-purpose simulation languages that can be applied to varied circumstances. A study done in 1983 stated that of

the simulations written, 40% were in GPSS, 19% in Simscript and 6% in SLAM. Another language, which offers a somewhat different and more promising approach, is Simula. Other languages or simulation packages are described briefly in the Appendix.

GPSS (General Purpose Simulation System), first released in 1961, was originally an internal IBM product used for computer performance modeling. A, newer, improved version, GPSS/H, has been released by Wolverine Software (Grain 1987).

GPSS is a block type of process oriented simulation language in which active entities or transactions move through sets of passive entities or blocks. A block diagram is constructed from many building blocks provided to describe the system. Using GPSS can lead to fairly easy to construct, succinct programs and easy model manipulations. If the system to be modeled fits the blocks provided, it is possible to construct a simulation very quickly.

However, many disadvantages exist. GPSS provides limited computational capabilities and poor facilities for generating random numbers so that it produces only approximate results. A user cannot ask for a number from the common probability functions; instead, any function must be given in tabular form. Because GPSS is integer based, discrete approximations for continuous values are sometimes necessary. It is an inflexible, closed package with no subroutines or extension mechanisms to allow for modifications. The input facilities are awkward and inflexible; it is difficult to input a variety of data forms. As the model size increases, many additional function, storage and initial statements are needed.

Statistics gathering is easy if you want the masses of automatically collected data. If you need other statistics, they can be quite difficult to obtain; it requires using more complex procedures with Help blocks to pull in Fortran code. Also, commands to gather statistics are spread throughout the program, making it harder to specify which are to be accumulated. Transactions are held on current events chains or future events chains in an inherently sequential fashion, making concurrency very difficult. The format of identifiers is rigidly structured. Because resources are passive, it is more difficult to describe complex systems. GPSS also lacks good control structures and algorithmic capabilities.

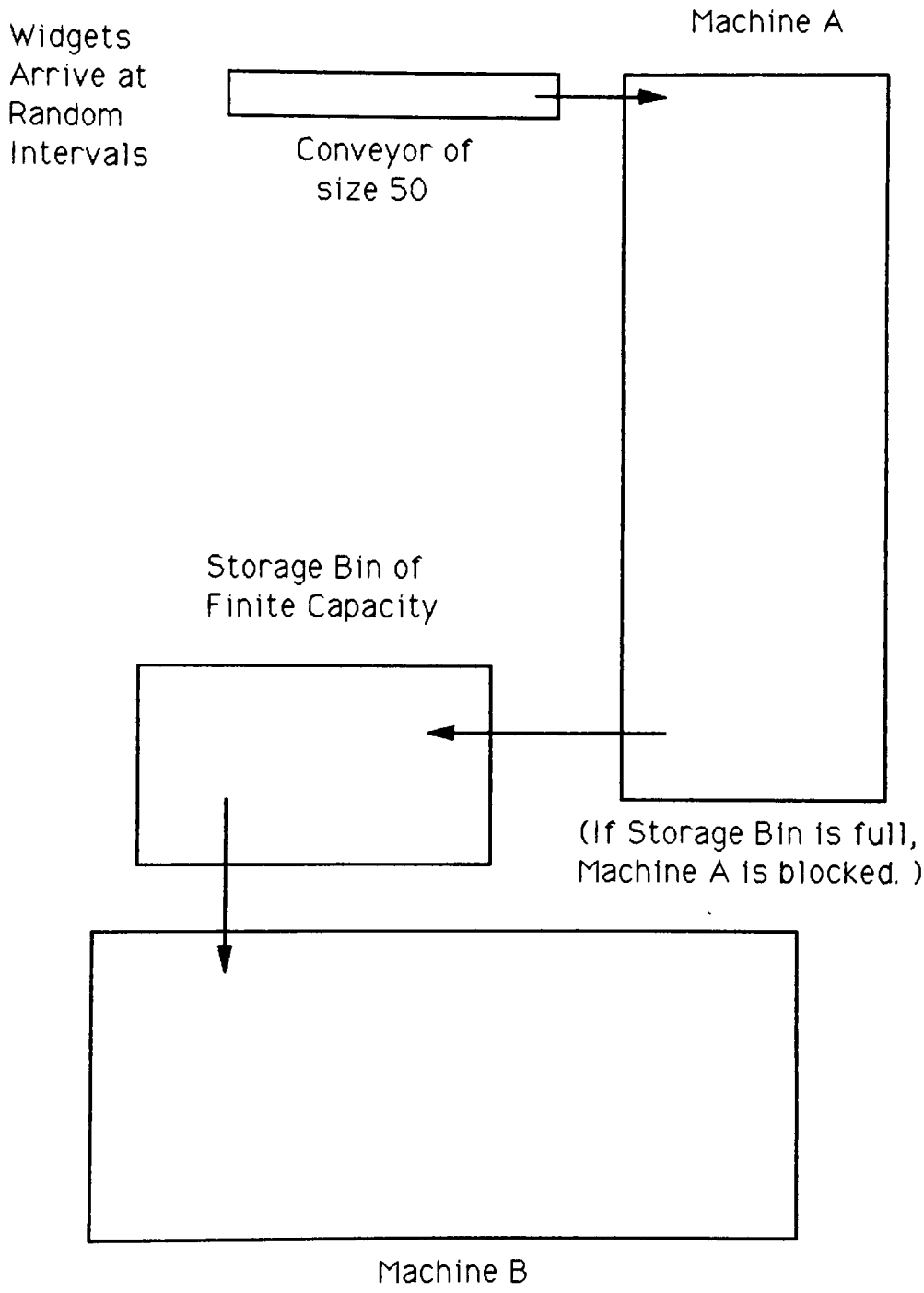
The newer version, GPSS/H, has interactive debugging facilities and improvements to the random number generators, allows calls to Fortran routines, and is much faster. Comparing the same representative programs written in GPSS, Simscript and SLAM, the GPSS/H programs compiled and executed much faster (Abed, 1985b). In order to provide more versatile control structures, additional general purpose programming statements were added such as DO-END, IF-ELSE and GOTO which seem of questionable value in making an understandable simulation language.

A classic widget manufacturing simulation could illustrate the varied features of GPSS, Simscript and SLAM. Assume widgets arrive randomly by conveyor to Machine A at a given arrival rate; the conveyor can hold 50 widgets; when the conveyor is full, widgets are sent elsewhere. After processing at Machine A, a widget is placed in a bin of size

C and fed into Machine B. If the bin is full, then Machine A becomes blocked. Machine B is subject to random failure; when it fails, the widget is removed and discarded (Banks, 1985). Figure 1 graphically illustrates this problem.

Figure 1:

Widget Simulation



A GPSS program to solve the widget problem is given in Figure 2.

Initially, variables are assigned values. Two storage resources are established, holdarea and conveyor, and assigned sizes. In addition, there are two facilities, MachineA and MachineB, which can process only one widget or transaction at a time. The generate command creates transactions using a call to a Fortran random routine (EXPON(&SEED1)) and the terminate command destroys them. The transactions move through the blocks of the program, entering and leaving the conveyor and hold area, and waiting to seize, enter, advance a set time for processing and release MachineA and MachineB. The runt block takes care of the random breakdown in MachineB.

Figure 2 : Widget Example in GPSS

```
   GPSS/H   VM/370 RELEASE 1.0

          SIMULATE
          INITIAL          XL1,59,/X2,40/X3,36
          INITIAL          XL4,8000,/X5,2000,/X6,1000

          STORAGE          S$HOLDAREA,40
          STORAGE          S$CONVEYOR,50
*
*         Time unit is 1/1000 of a minute
*
          EXTERNAL         &EXPON

          INTEGER          &SEED1,&SEED2
          LET              &SEED1=123456
          LET              &SEED2=234567

*         Create the widget transaction

          GENERATE         XL1*&EXPON(&SEED1)
          TEST GE          R$CONVEYOR,1,ASIDE
          ENTER            CONVEYOR
          SEIZE            MACHINEA
          LEAVE            CONVEYOR
          ADVANCE          X2
          FUNAVAIL         MACHINEA
          ENTER            HOLDAREA
          FAVAIL           MACHINEA
          RELEASE          MACHINEA
*
          SEIZE            MACHINEB
          LEAVE            HOLDAREA
          ADVANCE          X3
          RELEASE          MACHINEB
          TERMINATE
*
 ASIDE    TERMINATE
*
*         Create goblin transaction
*
          GENERATE         ,,,1
 RUNT     ADVANCE          XL4*&EXPON(&SEED2)
          FUNAVAIL         MACHINEB,RE,COUNT
          ADVANCE          X5,X6
          FAVAIL           MACHINEB
          TRANSFER         ,RUNT
*
          COUNT    TERMINATE
*
          GENERATE         200000
          TERMINATE        1
*
          START            1
          END
```

(taken from Banks, 1985)

Simscript is a more general purpose, flexible language than GPSS. It was originally developed as an event-oriented simulation system; later, a process-oriented modeling framework was added. According to the world view of Simscript, the parts of a system are divided into entities, processes and resources. Entities have attributes and may be grouped together into sets, for example, to represent an ordered list. The dynamic parts of the system are modeled by processes, which may follow such commands as work, wait, activate, request resources, or resume. Simple and efficient simulation control is provided by an event list and event notices. The language can handle externally generated events, provides adequate tracing and good random number generators, and is a good tool for scientific computing and list processing. It provides more readable code, and programs can be written in English-like statements. Simscript does not provide automatic statistics gathering and reporting like GPSS. However, it does allow a user to choose selectively which variables to monitor and about which to gather statistics. It can use the constructs tally and accumulate to collect data as needed. English-like statements such as 'Tally M as the Mean and V as the Variance of X' or 'For each man, list Average.Cash(Man) and Maximum.Cash(Man)' direct the gathering of statistics. In contrast to GPSS and SLAM, statistics may be gathered by adding commands only to the preamble of the program, without changing the main code. Simscript's event-oriented framework is more flexible than the process-oriented world view of GPSS and

is more capable of simulating complex systems. Simscript's read statement allows more powerful and flexible input of a wide variety of various data forms than in GPSS or SLAM. Animation is available on the personal computer version of Simscript through SimAnimation. Simscript is quite comprehensive and in common use (Russell 1987).

One of Simscript's main disadvantages is the accumulated layers of changes piled upon the original base, with the goal of maintaining compatibility with older versions. Furthermore, in contrast to GPSS's rigidity, Simscript is so flexible that it is easy to lose control. For example, there are no reserved words; everything can be redefined. There are many ambiguous constructs and inconsistent key words. For example, the two statements, 'ROUTINE EXAMPLE GIVING X' and 'ROUTINE EXAMPLE YIELDING X' have opposite meanings; in the first case, X is an input parameter and, in the second case, an output parameter. The uncontrolled syntax makes it harder to get the program correct. In addition, there is little possibility for concurrent execution, partly because of the need for a central event list. In comparison with GPSS/H and SLAM, Simscript is slower than GPSS/H but faster than SLAM for small and medium size models over short time periods and for large models generally (Abed 1985b).

A Simscript program to solve the widget problem is given in Figure 3.

The three major parts of any Simscript program are a preamble, main program and declaration of processes. The preamble declares the resources or fixed parts of the

Figure 3: Widget Example in Simscript

```
PREAMBLE
     PROCESSES INCLUDE ARRIVAL, WIDGET, BREAKDOWN, AND STOP.SIM
     RESOURCES INCLUDE MACHINE AND HOLD.AREA
     DEFINE M, A, B, LAMBDA, MU.A, MU.B
           AS REAL VARIABLES
     DEFINE JUNKHEAP, TOTAL, MACH.B.WIDGET AS
           INTEGER VARIABLES
     ACCUMULATE AVE.QUEUE AS THE AVERAGE AND
           MAX.QUEUE AS THE MAXIMUM OF N.Q.MACHINE
END

MAIN
     CREATE EVERY MACHINE(2)
     CREATE EVERY HOLD.AREA(1)
     FOR EVERY MACHINE
           LET U.MACHINE = 1      '' CAPACITY OF EACH MACHINE
     LET U.HOLD.AREA(1) = 40      '' CAPACITY OF HOLDING AREA

     LET LAMBDA = 17.0            '' PER MINUTE
     LET MU.A = 25.0              '' PER MINUTE
     LET MU.B = 28.0              '' PER MINUTE
     LET M = 8.0                  '' MINUTES
     LET A = 1.0                  '' MINUTES
     LET B = 3.0                  '' MINUTES
     LET TOTAL = 0                '' COUNTER FOR WIDGETS PROCESSED
     LET MACH.B.WIDGET = 0        '' POINTER TO WIDGET IN MACHINEB

     ACTIVATE AN ARRIVAL NOW
     ACTIVATE A BREAKDOWN IN EXPONENTIAL.F(M,1) MINUTES
     ACTIVATE A STOP.SIM IN 120 MINUTES
     START SIMULATION
END

PROCESS ARRIVAL
     IF N.Q.MACHINE(1) < 50
     ACTIVATE A WIDGET NOW
     ALWAYS
     ACTIVATE AN ARRIVAL IN EXPONENTIAL,F(1/LAMDA,2) MINUTES
END

PROCESS WIDGET
     REQUEST 1 MACHINE(1)
     WORK 1 / MU.A MINUTES
     REQUEST I UNIT OF HOLD.AREA(1)
     RELINQUISH 1 MACHINE(1)

     REQUEST 1 MACHINE(2)
         RELINQUISH I UNIT OF HOLD.AREA(1)
         LET MACH.B.WIDGET = WIDGET
         WORK 1 / MU.B MINUTES
         LET MACH.B.WIDGET = 0
     RELINQUISH 1 MACHINE(2)
     ADD 1 TO TOTAL
END
```

```
PROCESS BREAKDOWN
     IF N.X.MACHINE(2) = 1     '' MACHINE B IS BUSY
         INTERRUPT WIDGET CALLED MACH.B.WIDGET
         LET TIME.A(MACH.B.WIDGET) = 0.
         WORK UNIFORM.F(A,B,3) MINUTES
         RESUME WIDGET CALLED MACH.B.WIDGET
         ADD 1 TO JUNKHEAP
     ELSE                      '' MACHINE B IS IDLE
         REQUEST 1 MACHINE(2)
         WORK UNIFORM.F(A,B,3) MINUTES
         RELINQUISH 1 MACHINE(2)
     ALWAYS
     ACTIVATE A BREAKDOWN IN EXPONENTIAL.F(M,1) MINUTES
END

PROCESS STOP.SIM
     FOR EVERY MACHINE
     PRINT 3 LINES WITH MACHINE, AVG.QUEUE, MACHINE
         AND MAX.QUEUE THUS
     AVERAGE QUEUE CONTENTS FOR MACHINE * WAS ***.**
     MAXIMUM QUEUE CONTENTS FOR MACHINE * WAS ***

     PRINT 2 LINES WITH TOTAL-JUNKHEAP AND JUNKHEAP THUS
     TOTAL NUMBER OF GOOD WIDGETS PROCESSED WAS *****
     NUMBER OF WIDGETS DISCARDED WAS             *****

     STOP
END
```

(taken from Banks, 1985)

problem. A machine resource is declared and also a hold.area for the first machine. The conveyor belt does not need to be declared because all resources have associated queues. In the main program, the resources are created and assigned capacities, variables are assigned to be used in the random functions and processes are activated. The key agents are the four processes which are listed in the preamble and then further defined below. Process arrival creates the widgets according to a random function and then process widget contains the code for the actions the widget will perform. Process breakdown handles the random breakdown of the second machine. The desired output is listed under process stop.simulation. Notice the code is much more understandable than in GPSS.

Fortran has been used to write simulation programs for many years. The main advantage of Fortran is that it is well-standardized and available. However, it has no adequate means for defining and manipulating data structures and is not as powerful, elegant or easy to use as other languages.

Several packages of subroutines to meet the most likely needs of simulation users have been written to be used with Fortran. GASP II consists of about two dozen routines to transparently manipulate data structures. Building upon GASP II, SLAM was initially released in 1981 (O'Reilly 1987). Its process-oriented framework uses a network model made up of nodes and branches to represent the elements in the process. The model is built by combining nodes, which represent locations where processing takes place, and branches, which define movement of entities through the model. SLAM supports process interaction, event scheduling and continuous modeling perspectives. However, adding such additional features interferes with the clarity of the model. SLAM has better input specifications than GPSS but is not as flexible or powerful as Simscript. Combined with TESS, it allows graphical construction of the SLAM network model, translation of graphical input to statement form, database management, output analysis and scenario animation, multiple displays, and interactive debugging. In comparison with GPSS/H and Simscript, SLAM is slower than GPSS/H but faster than Simscript for small and medium models for over long time periods (Abed 1985b).

A SLAM program to solve the widget problem is given in Figure 4.

Figure 4: Widget Example in SLAM

```
NETWORK;
:
          RESOURCE/MACHB(1),3,2;
:
:    WIDGET PROCESSING
:
          CREATE,EXPON(XX(1));
          QUEUE(1),,50,BALK(LOST);
            ACTIVITY/1,XX(2);
:
          AWAIT(2/40),MACHB/1,BLOCK;
            ACTIVITY/2,XX(3);
          FREE,MACHB;
          TERM;
:
LOST GOON;
          ACTIVITY/5;
          TERM;
:
:  FAILURE SUBNETWORK
:
          CREATE;
            ACTIVITY.EXPON(XX(4));
RUNT      PREEMPT(3),MACHB.COUNT;
            ACTIVITY/3,UNFRM(XX(5),XX(6));
          FREE,MACHB;
            ACTIVITY.EXPON(XX(4)),,RUNT;
:
:  COUNT THE BAD WIDGETS
:
COUNT GOON
          ACTIVITY/4;
          TERM;
          ENDNETWORK;
:
INIT,0,120;
INTLC,XX(1) = .0588,XX(2) = .04,XX(3)= .0357;
INTLC,XX(4) = 8.,XX(5) = 1.,XX(6)= 3.;
FIN;
```

(taken from Banks, 1985)

The SLAM program is more concise but more confusing and harder to understand than the Simscript program. Each machine is represented by a branch or activity statement, preceded by a statement giving the file where entities will wait if need be. Entities are created by the create node using the random function (EXPON(XX(1))). Machine A is simulated by Activity/1. Its holding area is specified by the Queue node as file 1 with size 50. If it's full, the entities go to the lost node and Activity/5. Processing in Machine A takes XX(2) time. Machine B is declared as a resource with two files (3,2) associated with it. Upon leaving Activity/1, entities attempt to enter Machine B (Activity/2). The Await node represents the conveyor belt or file 2 of size 40. If this file is full, Machine A becomes blocked. If an entity successfully enters Machine B, it is processed for XX(3) time, frees the machine and is terminated or destroyed. The failure subnetwork takes care of the random failure, modeled as Activity/3. Statistics are automatically accumulated about each activity.

Simula, developed at the Norwegian Computing Center starting in 1963, is built around processes. The system to be modeled is described in terms of how each process within the system views and interacts with the rest of the world. This can lead to easier updating since if a process changes, just the module for that process needs to be modified. Simula implements the elegant and natural concept of classes, which provide powerful facilities for structuring data. A class is a set of objects with similar characteristics; an object is a structured variable that can contain data fields and instructions or routines to be executed; such objects can be created and destroyed dynamically. Each object inherits from the class of which it is a part. Furthermore, classes may inherit from superclasses. Objects might aid in the implementation of parallel processing, where each object has its own processor. Because of the division of the system into classes and objects, the program can be easily modified with undesirable features deleted and better structures added.

However, there are disadvantages of Simula. Bratley asserts that it contains some excellent ideas rather badly carried out (Bratley 1987). It seems unnecessarily complex and requires a working knowledge of Algol. Also, there is a tedious way of accumulating statistics and poor formatting of reports.

In order to make Simula more workable, Demos was built on top of Simula providing some GPSS-like utilities to make it easier to use. Demos includes Simula objects, which strongly

resemble GPSS transactions; a Demos resource is similar to a GPSS storage. Demos combines the ease of use of predefined blocks and entities in GPSS with the extensibility and power of Simula (Bratley 1987). It also includes automatic report creation. Both Simula and Demos are being used more in Europe than in America.

For a more concise analysis of these languages and their ability to fulfill the requirements for a good simulation language, see Table 1.

Table 1

	GPSS	Simscript	SLAM	Simula
Concise and comprehensible representation of real world entities	-	-	-	+
User-friendly, unambiguous, English-like syntax	-	?	-	?
Clear and well-defined semantics for user	-	-	-	-
Easy-to-use powerful data structures	-	+	?	+
Efficient, modifiable clock	-	?	-	+
Many good, random number generators	-	+	+	+
Reasonably efficient (capacity for concurrency)	?	-	-	?
Capacity to collect, print and analyze statistics	-	?	+	?
Full interactive debugging	+	?	+	-
Easy modification - need for structural modularity, capacity for scaling	-	-	-	+
Capacity for intelligent exploration	-	-	?	-
Graphics and animation	?	?	+	-

Key: + Fulfills requirement
? Fulfills part of requirement
- Does not fulfill requirement

2.3 Object-oriented Approach to Simulation

A programming paradigm that seems better able to meet the requirements for a simulation language is object-oriented programming. Let us look at what the object-oriented paradigm entails and how it can be used for simulation.

The object-oriented approach traces its roots to Simula and the concepts of classes and objects. Object-oriented languages subdivide a system into objects, that is, integrated units of data and procedures acting on the data. Such procedures are called methods. The value of an object's data, its instance variables, represent the state of the object. The only way that one object may interact with another is by sending a message requesting that the receiving object execute one of its methods. Because of this, objects encapsulate their data, i.e. serve as units of protection where each object can guard its data against external actions that may make the internal data inconsistent. Objects are dynamic and can be created during the execution of the program. Objects are grouped into classes which describe the behavior of a kind of object, an instance of the class. This description includes the nature of the internal data and the methods that can be executed. Subclasses may inherit the structure and methods of a class. Object-oriented classes are polymorphic; i.e. the same message can be sent to both a class and its subclasses. For example, if there was a polyhedra class, and cube, prism and tetrahedron were subclasses, all the classes could receive a standard message 'print volume' and respond correctly according to the appropriate method for their geometry.

Object-oriented languages could fulfill many of the requirements for a good simulation language.

Objects are grouped together in hierarchical classes which can then be put into separate modules. Parts of the program outside a given class or module can interact only in specified ways with the class or module and thus do not need to know how given methods are executed, variables are changed or classes are structured. More primitive ideas can be encapsulated in the super classes, thus reducing the level of complexity visible in the subclasses. Information about a particular structure or implementation can be hidden within an object or class.

Such a modular structure can make modification easy since one module can be altered without affecting others. This can lead to more flexible and extensible programs. It encourages software reuse; modules from different simulation programs can be pulled in when constructing a new program. A user can easily make "on the fly" changes; new methods can be defined or new classes and objects added. Subroutines and utilities from other languages can also be utilized. This modularity gives the user the ability to pull desirable features into an object-oriented simulation program such as a better random number generator or structures to gather statistics and print reports. Large programs can be broken up into many small, independently functioning, units.

Programs in object-oriented languages can be readable and comprehensible with English-like statements, intelligible to non-programmers. Such programs can clearly illustrate which entities interact and how, and verify how well the model

reflects the real world. The capacity for multi-level reading of simulation programs, focusing on different levels of the class hierarchy or modules, can give new users a quick overview or a more experienced user an in-depth look at a specific module.

The use of messages and objects seems a natural way to model many interactions. This style of programming parallels the way we intuitively think of processes in dynamic systems. Behaviors are attached to specific objects just as real world entities exhibit different behaviors. Such a software design can also correlate programming objects on a one-to-one basis with real world objects. It specifies in one place all the data associated with an object and the routines or methods that can manipulate that data. Such structure can allow both naive and experienced users to understand quickly a model.

The object-oriented design also appears to be well-suited for distributed environments or concurrent execution. Objects or modules can be placed on different processors and communicate via message passing. Distributed designs such as Time Warp and TimeLock are being explored to see how different behaviors could be synchronized in such an environment.

The interactive nature of many object-oriented languages also tends toward intelligent exploration. The program could be interrupted while running (in contrast to the compiled nature of Simscript) and the state queried or code modified, and then the simulation resumed. Alternative designs could be explored more quickly and easily, and debugging could be

simpler and faster.

Examining the details of object-oriented languages will make these advantages clearer.

Smalltalk is an object-oriented language that has been used to construct simulations. It uses an event-driven model employing objects, that is, instances of different classes, to represent the entities of the model. Class SimulationObject represents any kind of object that enters into a simulation in order to carry out tasks. Class Simulation represents the simulation itself and provides control structures for admitting and assigning tasks to new SimulationObjects. Objects operate more or less independently so activities are synchronized by instances of class Semaphore and Shared Queue.

Smalltalk has an excellent tool environment, with many available resources and options. Statistics about throughput, tallying and monitoring events, and duration can be gathered and printed using a variety of output forms such as histograms or graphs. Smalltalk implements the client-server model in which there are both active and passive objects. For example, in modeling a crossroads, cars would be active while the intersection would be passive. Such a model is more natural, has less of a chance of deadlock and can use inheritance effectively. For example, all vehicles could be of a common class with subclasses for trucks, police cars and cars, where objects in subclasses could inherit certain common traits but also have their own specialized features.

On the negative side, Smalltalk utilizes many small procedures, which can result in a very high collective

procedure call overhead. It is also somewhat deficient in the encapsulation of instance variables in that it allows free access to inherited instance variables by descendant classes. It also doesn't provide the performance and static checking of other object-oriented languages.

SimTalk, a simulation package written in Smalltalk, provides queuing support, statistics gathering, simulation-oriented graphics and an interactive user interface, which makes modifying running simulations easier. It provides a large number of predefined objects, a tracing facility and the ability to suspend a simulation, change parameters, and then continue. SimTalk simulations have been written to model such systems as a manufacturing plant, a complex computer architecture, and a distributed database network (Knapp, 1987).

Ross is an event-based, object-oriented, simulation language developed by the Rand Corporation and implemented in Lisp. Objects are organized in a hierarchy of class-subclass links. An object automatically inherits attributes and behaviors of classes to which the object belongs. Ross also includes specialized objects such as The Physicist or The Mathematician which have knowledge on how to query the state of the simulation. Message passing is implemented using pattern matching. In contrast to other object-oriented languages, each Ross object has its own list of tasks and associated execution times; the clock implementation also includes the list of the next action for each active object in the simulation; the clock sends messages to each object when they should act (Klahr, 1985).

In Ross, the user can write abbreviations, which can significantly enhance code readability, for example, 'ask myself'. Since it allows a human to play the role of one of the objects in the simulation, Ross provides a good environment for training and analysis. The English-like code is easy to use, readily understandable and easy to modify.

Two large simulations have been written in Ross: SWIRL (Strategic Warfare in the Ross Language) which simulates offensive and defensive air battle tactics and TWIRL (Tactical Warfare in the Ross Language) which models ground combat. Color graphics are used to show the progress of the combat.

It would be difficult to run Ross concurrently because of the event queue and global clock.

C++ is an object-oriented language based on the C language. It was developed at Bell Labs in the early 1980's by Bjarne Stroustrup to aid in writing complex discrete event simulations. It supports data abstraction, encapsulation and polymorphism. It has strong type checking, auto construction/ destruction and function and operator overloading. Depending on a programmer's purposes, either the object-oriented paradigm or procedural constructs can predominate. C++ is becoming more and more widely distributed with versions available from Zortech, Oasys, Guidelines and Oregon. AT&T is rewriting the Unix operating system in C++. According to Weiner, C++ combines the advantages of object-oriented programming with C's power of expression, low level system programming, efficiency, and economy (Weiner, 1988).

One of the object-oriented extensions to Common Lisp is

Flavors. A flavor or class is an abstraction of the characteristics that all objects of this flavor have in common. The term generic function stands for an operation on an object. A method is a Lisp function that performs a generic function on instances of a certain flavor.

Flavors has many advantages. Generic functions serve as interfaces between objects to provide abstraction and isolation between modules. Multiple inheritance allows object types to be built up from a toolkit of component parts. A typical flavor is defined by combining several other flavors or mixins. A new flavor inherits instance variables, methods, and additional component flavors. A user can easily define new method combination types. Macros are provided that accept a declarative specification of method sorting, filtering and combination, and automatically produce detailed code to combine methods. Ordinary and generic functions are called with the same syntax so the caller doesn't need to know which kind is involved. A wide variety of tools is available for analyzing Flavors-based programs and for inspecting the current state of the system. In addition, Flavors offers a lot of flexibility in redefining parts of a program. The programmer can redefine flavors, methods and generic functions at any time, even while the program is running. When a flavor is changed, the system propagates changes to any flavors of which it is a direct or indirect component. Flavors includes full runtime error checking. Also, data hiding is implemented; a user can employ a function without having to know about the inner working of that function. The functionality of a program can be changed by

adding new code rather than by modifying what's already there.

A disadvantage of Flavors is that it does not enforce the convention that only an instance's methods can access a given instance variable; instead, it allows free access to inherited instance variables by descendant classes. Thus, full encapsulation of instance variables isn't implemented.

2.4 Conclusion

Object-oriented languages hold great promise for use in simulation. The potential to meet the requirements of a good simulation language is much greater than for traditional procedural or simulation languages. If we look again at the requirements for a good simulation language, we can see that object-oriented languages could fulfill all of them (See Table 2.). As we have seen, object-oriented languages allow concise and comprehensible representation of real world entities as objects with certain behaviors and values and can provide user-friendly, English-like code with clear and well-defined semantics. The hierarchical structure of classes and subclasses provides a powerful way to organize and structure data. Because of the modular nature of object-oriented languages, many desirable features of a simulation language such as an efficient time mechanism and good random number generators can be provided. Object-oriented languages have the capability for distributed execution where different objects could be placed on different processors and run their methods in parallel. Most of the available object-oriented languages have interactive debugging, the ability to suspend and restart a simulation, and graphics and animation. Most

Table 2

	GPSS	Simscript	SLAM	Simula	Object-Oriented Simulation
Concise and comprehensible representation of real world entities	-	-	-	+	+
User-friendly, unambiguous, English-like syntax	-	?	-	?	+
Clear and well-defined semantics for user	-	-	-	-	+
Easy-to-use powerful data structures	-	+	?	+	+
Efficient, modifiable clock	-	?	-	+	+
Many good, random number generators	-	+	+	+	+
Reasonably efficient (capacity for concurrency)	?	-	-	?	+
Capacity to collect, print and analyze statistics	-	?	+	?	+
Full interactive debugging	+	?	+	-	+
Easy modification - need for structural modularity, capacity for scaling	-	-	-	+	+
Capacity for intelligent exploration	-	-	?	-	+
Graphics and animation	?	?	+	-	+

Key: + Fulfills requirement
? Fulfills part of requirement
- Does not fulfill requirement

importantly, since object-oriented programs can be easily modified with new objects, classes and modules added as needed, as requirements change or new demands are made of a simulation, new modules may be constructed without having to change the whole program.

Object-oriented languages are much more able than traditional languages to meet the growing need for effective simulation and to provide a powerful tool to aid users in understanding and predicting a system's structure and behavior.

Chapter 3 - Distributed Simulation

A facet of object-oriented simulation that is particularly intriguing and promising for the future is its capacity for distributing execution. However, some problems have been encountered when switching from sequential to distributed simulation. Proposed solutions may lead to elegant and insightful algorithms or, if not carefully done, further entanglements.

Two main components of sequential simulation are the event list and clock, both of which are inherently sequential. How could the responsibilities of the event list and clock be handled with distributed discrete event simulation? With parts of the simulation running on different processors, how would the processes be synchronized so that the correct chronology of actions will take place?

A real world system could be viewed as having many interacting physical processes. In a discrete event simulation, each physical process is represented by a logical process. Logical processes, placed on different processors, could communicate via messages with a time stamp representing when it was sent and/or received. The communicating logical processes could be pictured as a directed graph with edges between the nodes or logical processes that have message sending/receiving capacity. The different logical processes could be represented as objects residing on different processors and having the capability to pursue their own tasks independent of the other processes. They would communicate by sending messages, but it would be vital

for the validity of the simulation that the messages be sent and acted upon in chronological order. In other words, the behavior of a process at time T cannot be affected by any information sent to it after time T.

One possible approach is tight synchronous simulation, where there is a global clock storing the same time for all the processors; all the processors would have to agree when the clock is to be updated. This approach would have little potential for speedup over the sequential option; in fact, with the additional message passing among processes, it would probably be slower.

In order to benefit from having processes run on different processors, processors have to be allowed to maintain their own times and proceed autonomously with their actions. However, limits have to be put on how far forward a process can go so that it doesn't violate the laws of causality, i.e. an action occurs in the present, which should have occurred in the past and changed current conditions. The possibility of such 'time faults' is the key problem of loose asynchronous discrete event simulation. Furthermore, processes may temporarily wait for an incoming message with some needed data. This introduces the possibility of deadlock while all processes are waiting or blocked. In the simplest case, process1 is waiting for a message from process2, which is waiting for a message from process1. Such deadlock is not unlike what occurs in operating systems and databases.

Three main approaches to solve the problems of loose asynchronous discrete event simulation are:

1) deadlock avoidance -structure the system so deadlock cannot occur

2) deadlock detection and recovery - provide means to recognize when deadlock has occurred and recover from it

3) optimistic with state rollback - have each process charge ahead and if a message appears from its 'past', rollback to that time and recalculate.

The main proponents of the first two approaches are Chandy and Misra (1978, 1979, 1981, 1986). Their original strategy (1979) was to avoid deadlock. The main vehicle was a no-job or null message telling a process that no message would be coming until a given time. Thus, the receiving process could safely proceed with any required actions until that time. This could result in a very high overhead of null to real messages (Seethalakshmi, 1979). A slight modification suggested by Chandy and Misra was utilizing demand driven null messages, an arrangement where a process would only send a null message when a receiver requested it. Other alternatives could be a circulating marker scheme, where a single marker carries all null messages and also gathers simulation statistics, or a time out mechanism, which delays sending a null message for an interval, anticipating that a regular message would be sent soon.

In 1981, Chandy and Misra proposed, instead, that deadlock be detected and then recovery made. They suggested that a processor set a flag in global memory when it thinks it's deadlocked; a guardian processor would check whether all flags were set, i.e. all processors thought they were

deadlocked; if so, it would send a message to each processor to check that it was really deadlocked. In order to recover, a special process would determine the minimum of the next scheduled time for a local event at any node and send messages to advance all local times to that minimum (Chandy and Misra, 1981). This detection and recovery would require additional software and time.

Evaluations of these methods have varied. Work by Peacock et al (1979), Seethalalshmi (1979), and Quinlivan (1981) have shown that simulation performance can be improved substantially by the methods outlined above. However, Reed actually tried the Chandy-Misra approach on a multiprocessor and concluded that it was not viable for queuing network problems because of the excessive overhead for message passing. For example, with a central server network model using deadlock avoidance, about twenty null messages were sent for each single valid message (Reed, 1988).

The third approach, optimistic asynchronous simulation with rollback, was first exemplified by the Time Warp mechanism, developed by Jefferson (Jefferson 1985c). With this strategy, each process continues pursuing its own tasks until a message arrives that needed to be processed in the process's 'past', i.e. with a time less that the processor's current time. The processor then rolls back until before the time of the message, sends out anti-messages cancelling any messages it might have sent in the intervening time, and then moves ahead from the time to which it rolled back. The anti-messages may, in turn, cause other processors to roll back.

With this approach, each process has a name, a local virtual clock maintaining the processor's local time, and a state, which consists of its execution stack, its own variables and its program counter. It also has a state queue, storing saved copies of the process's most recent states with at least one state older than the global virtual time of the simulation. An input queue stores all received messages ordered by their time stamp. An output queue contains negative copies (an identical message preceded by a -) of all messages sent out so that if rollback is necessary, anti-messages can be sent. Each message includes the sender's name, the virtual send time, the receiver's name, and the virtual receive time.

A process moves ahead, processing the messages in its input queue with the lowest receive time, gambling that no message or straggler with a time stamp less than its local virtual time (LVT) will arrive. After processing a message, the LVT is increased to the time stamp of the next message in the input queue. If a straggler arrives, the process rolls back. It finds a state in its state queue before the receive time of the straggler, restores that state and the LVT of that state, and discards all newer states. It also moves the pointers in the input and output queues back to before the restored LVT. It sends all the anti-messages in its output queue between its restored LVT and its previous LVT, and begins execution again.

When an anti-message is sent, it may find the original message still in the receiver's input queue, whereupon both messages will be annihilated. The original message already

may have been processed so the receiver process will need to rollback also. A third possibility is that the anti-message may arrive before the original message; in that case, the anti-message will wait in the input queue for the original message and, then, will destroy the positive one when it arrives.

There are two strategies for determining rollback: aggressive and lazy. In aggressive cancellation, all messages are cancelled; in lazy cancellation, only those messages that are different are cancelled. To cancel all messages in aggressive cancellation, all the anti-messages in the output queue are sent. With lazy cancellation, the processor continues executing after its state has been restored. When a message is produced, it is compared to the anti-message in the output queue previously produced before rollback. The anti-message is only sent if the new message is different; in that case, the new message is also sent. If the new message is the same, no message is sent since a copy was sent previously. Studies have shown that lazy cancellation can be much faster than aggressive cancellation, especially for cases where the straggler does not affect the output queue, but it requires more memory since the output queues need to be saved after restoration rather than wiped out as they are with aggressive cancellation (Gagni 1988).

At regular intervals, a global virtual time (GVT) has to be determined in order to limit the size of state queues and allow valid output. This is determined by broadcasting a request to each processor asking for its LVT and the lowest virtual send time of all unprocessed messages in its input

queue, and then setting the GVT to the minimum of these. When a new GVT is determined, messages with a smaller time stamp in the input and output queues can be destroyed; only one state older than the GVT needs to be saved in each processor's state queue.

If a receiver's input queue is full, the 'newest' message, the one with the highest time stamp is returned to the sender, causing that processor to roll back and later try to re-send the message. Any irretrievable output can be done only when the time requesting it is older (less than) the GVT to assure that it won't be rolled back.

Jefferson supports his approach by stating that it's an "elegant and natural implementation" (Jefferson 1985c). Furthermore, the time spent rolling back isn't that significant, considering that, in other implementations, the processes would be deadlocked or blocked for that length of time. He further suggests that rollbacks should be quite infrequent; because of the temporal locality assumption, it is expected that most messages will arrive in the virtual future of the receiving object. The rollback mechanisms will keep processes with frequent communications relatively close in LVT and slow down very rapid processes by frequent rollbacks, bringing them closer to the GVT and thus preventing them from often causing slower processes to rollback (Jefferson, 1985c).

Time Warp is the focus of much research and experimentation. Numerous studies have shown that the Time Warp mechanism can result in significant speedups over the conservative asynchronous and sequential approaches. Lomov,

Cleary et al (1988) found that Time Warp using lazy cancellation 'can achieve good speedups on a large number of processors' using a Time Warp simulator written in C++. Berry (1986) asserted that Time Warp with lazy cancellation can perform better than the conservative synchronization approach. Gilmer (1988) concluded that Time Warp is a viable approach for distributed discrete event simulation with 'acceptable overhead and efficiency.' Madisetti et al (1988) suggested an alternative implementation for Time Warp, where the sphere of influence is limited and thus the rollback wave effects can be contained. Fujimoto et al (1988) states that Time Warp is superior to the Chandy and Misra approach because it avoids deadlock and blocking problems; however, he proposes a rollback chip to lessen the overhead associated with rollback.

In order to explore complex real life systems, distributed simulation will need to be utilized to allow results to be obtained in a reasonable time frame. However, distributed simulation introduces time synchronization problems. Two possible solutions are the conservative Chandy-Misra approach, where time in the simulation can be moved ahead only when it can be determined that all events up to that time have been executed, and the optimistic Time Warp, where each processor can move ahead as quickly as possible but strategies such as rollback are necessary to deal with possible time faults. More work is needed on these approaches to make distributed simulation a viable tool to simulate real life systems efficiently.

Moving from a discussion approach to actual implementation of distributed discrete event simulation, I helped develop a distributed simulation package running on a network of Texas Instrument Explorers at RIT Research Corporation. This package is an extension of an existing simulation package, in Flavors, built upon the ESPRIT Speech Research Environment, mainly written by Rob Gayvert, Principle Software Engineer. I then used this distributed simulation package to write a module or program based on the widget example in Chapter 2.

The major flavor in the original simulation package is 'simulation', with instance variables for objects, the clock, status of the simulation, domain or subject matter being simulated, and the console (if any) being used to record the events of the simulation. Another key element of the simulation package is the 'basic-object' class or flavor, which includes mixins to handle message passing, process handling, listing the actions taken by an object, time and graphical representation. The inheritance property of the basic object flavor was used to construct subclasses for dynamic objects (clients) and physical objects (servers). Clients are simulation entities, which are provided with a service, for example, customers, where servers provide a service, for example, a bank teller.

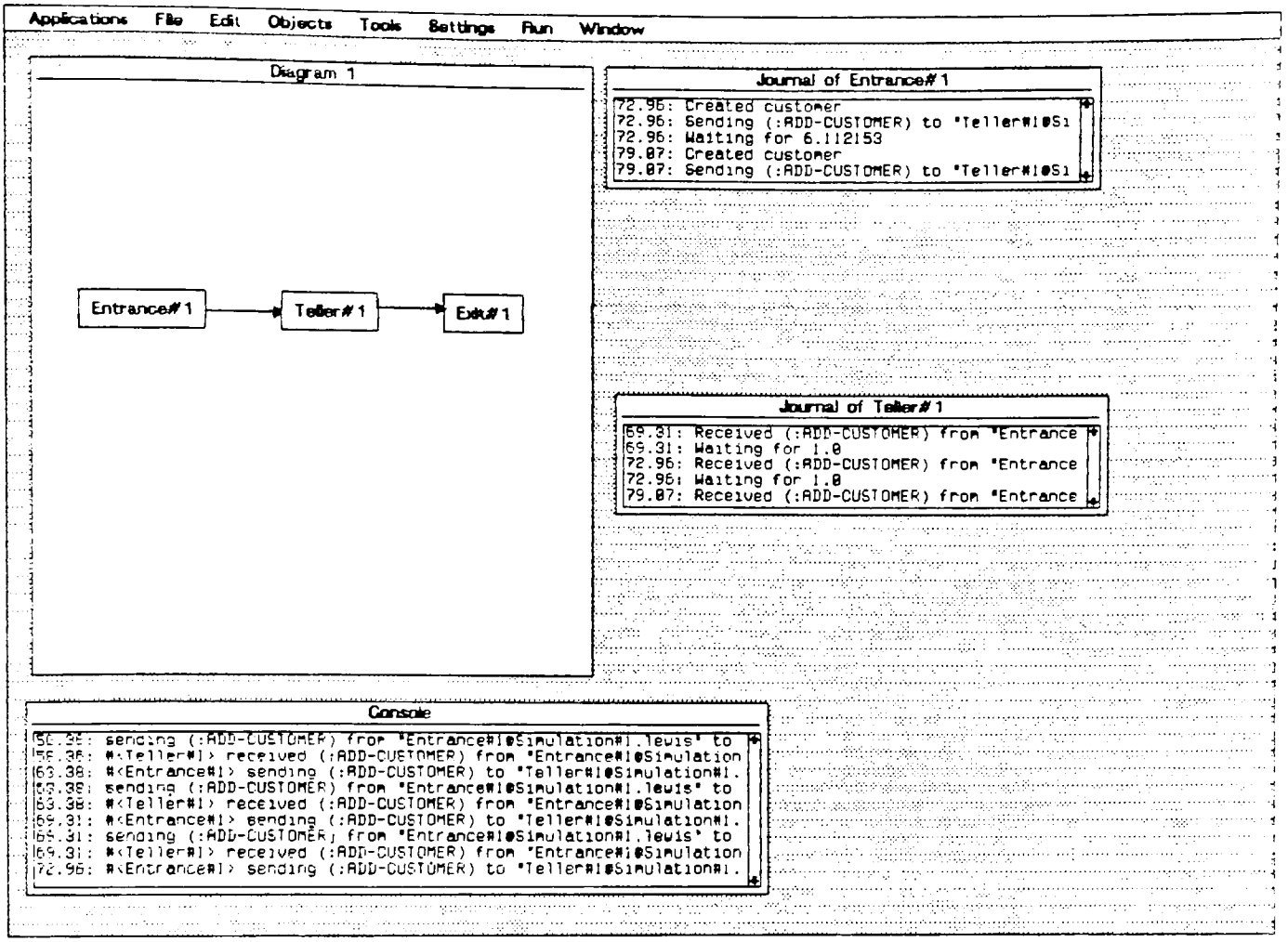
Using this platform, a user can graphically represent physical and dynamic objects from a simulation on the screen, run the simulation, and view the actions

through a journal or console. Each instance of an object is represented by a rectangle, and paths are drawn between appropriate objects to represent the real life system. For example, in Figure 5 an entrance, one teller, and an exit are drawn to simulate a bank. These objects are selected from the pop up menu Objects. They can then be moved around the diagram as desired. To connect the objects, the Tools menu is popped up and the crayon icon selected. Arrows are drawn indicating the paths of dynamic objects or customers. Double-clicking on an object will allow the setting of specific parameters, such as what actions a teller can perform or what random function will determine the creation of customers by entrance. In order to be able to see what is happening as the simulation executes, journals for objects and a console for the simulation can be added. By choosing the pop up menu Window, a user can select Console, which will report the actions of the simulation. In order to get a journal to report the actions of an individual object, such as Teller#1 in Figure 5, the object is highlighted and then Journal is selected from the Window menu. To run the simulation, select the Run menu and highlight Start. By using Pause and Continue in the Run menu, the simulation could be halted, modifications made and then resumed.

We wanted to convert this system so that it could be run in a distributed fashion; i.e. the different objects could be placed on remote networked Explorers.

To be able to this, information needed to be stored

Figure 5


Applications
File
Edit
Objects
Tools
Settings
Run
Window
Diagram 1
Entrance#1
Teller#1
Exit#1
Journal of Entrance#1
72.96: Created customer
72.96: Sending (:ADD-CUSTOMER) to "Teller#1@Si
72.96: Waiting for 6.112153
79.07: Created customer
79.07: Sending (:ADD-CUSTOMER) to "Teller#1@Si
Journal of Teller#1
69.31: Received (:ADD-CUSTOMER) from "Entrance
69.31: Waiting for 1.0
72.96: Received (:ADD-CUSTOMER) from "Entrance
72.96: Waiting for 1.0
79.07: Received (:ADD-CUSTOMER) from "Entrance
Console
63.38: sending (:ADD-CUSTOMER) from "Entrance#1@Simulation#1.lewis" to
63.38: #<Teller#1> received (:ADD-CUSTOMER) from "Entrance#1@Simulation
63.38: #<Entrance#1> sending (:ADD-CUSTOMER) to "Teller#1@Simulation#1.
63.38: sending (:ADD-CUSTOMER) from "Entrance#1@Simulation#1.lewis" to
63.38: #<Teller#1> received (:ADD-CUSTOMER) from "Entrance#1@Simulation
69.31: #<Entrance#1> sending (:ADD-CUSTOMER) to "Teller#1@Simulation#1.
69.31: sending (:ADD-CUSTOMER) from "Entrance#1@Simulation#1.lewis" to
69.31: #<Teller#1> received (:ADD-CUSTOMER) from "Entrance#1@Simulation
72.96: #<Entrance#1> sending (:ADD-CUSTOMER) to "Teller#1@Simulation#1.

about the main simulation and any remote simulations. Additional methods were added to the original simulation class to initialize remote simulations, add remote hosts, create remote connections through the Chaos communication network and make remote objects. Instance variables were added to keep track of a simulation's remote objects and hosts, contact paths for each simulation so that messages could be directed properly, and which Explorer was the residence of the main simulation. A subclass of simulation, remote-simulation, was established with separate instance variables for the main host and the remote host Explorer.

Methods were written to facilitate message passing between the main and remote simulations. Message handling was set up so that if a message was directed to an object on a remote machine, it was sent first to the local simulation, then to the remote simulation and then to the remote object. Contact paths to correctly direct the messages were constructed from object name, simulation name and host name, e.g. Machine#2@Remote-Simulation#3.Clark. Starting with the system's remote-eval function, which facilitates communication between different machines, we developed a simpler, unidirectional, more efficient method to allow better inter-machine communication.

In the original simulation package, dynamic objects or clients, like customers in a bank, were created by an entrance object and directed to an appropriate teller or server by a router object. However, objects cannot be sent directly over the network, so to use the same arrangement might require that dynamic objects be created multiple times

on many machines. In order to avoid this inefficiency, messages were used to represent dynamic objects and sent from server object to server object with the necessary information to simulate a client.

In the original simulation package, an instance of simulation class initially started all objects. When all objects had completed their processing for that time, the simulation checked the time of the next event for each object, moved the clock to the closest time of an event, and started the objects again. For the distributed simulation, each remote simulation sets a next-wakeup-time instance variable in its remote connection to the time of its next event. Then, in the run method for the main simulation, both local objects and remote connections are polled to find the smallest time any object has an event scheduled. The main host advances its clock to this time and sends a message to each remote simulation to wakeup and advance their clocks to this time. This represents the traditional, lockstep, conservative approach to synchronization.

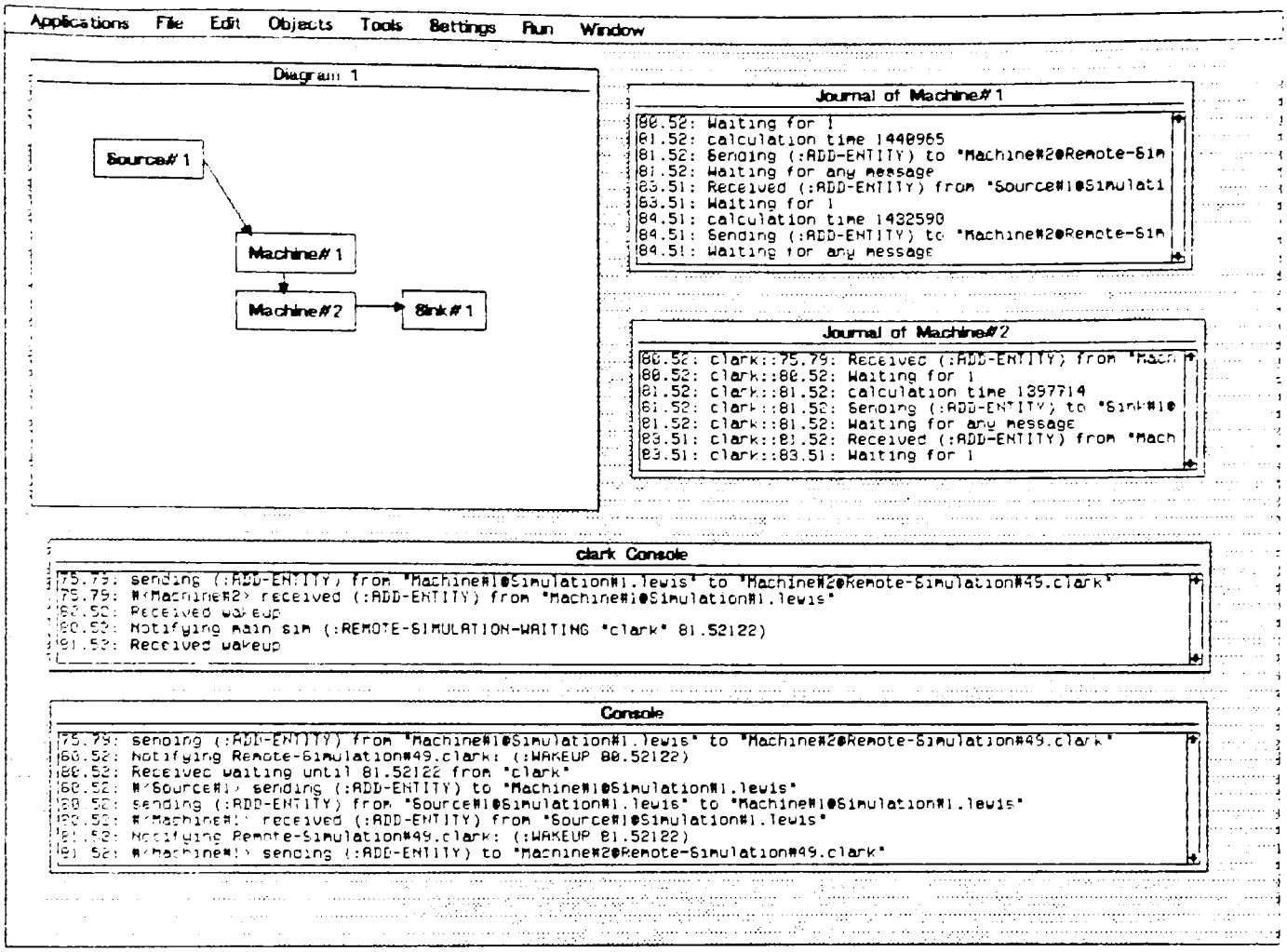
The main difficulty in getting the new distributed system to work was that it frequently hung up and would not progress past certain points. After much investigation and trial and error debugging attempts, it was discovered that the main problem was that the different processes were interfering with each other and apparently intermingling their messages in the network stream. This was taken care of by wrapping 'without interrupts' macros around strategic places in the code. Consoles for the remote host and journals for remote objects were useful debugging and reporting devices.

Sample simulations were run based on a bank and car wash. To test more fully the distributed package, and to be able to compare the design process, execution and output of distributed object-oriented simulation to traditional strategies, a variation on the classic widget manufacturing example, described in Chapter 2, was implemented with this package.

Classes, based on the basic object class, were created for a source, which creates widgets at random intervals, the actual machines, which do the processing, and a sink, which disposes of widgets and accumulates final statistics. Using pop up menus, a user can choose for each machine instance a variety of manufacturing processes (drilling, turning, milling, grinding, inspecting), a TI Explorer on which to place the operation of that machine, and the length of time the manufacturing process will take. For the source object, the user can choose which random function to use to determine the arrival of widgets and on which Explorer to place the source. Similarly, the sink can be placed on any computer.

Figure 6 illustrates these components of the simulation. The four objects were created by using the menu Objects and choosing which objects were desired. Journals for Machine#1 and Machine#2 and the console were chosen as described previously. To get a reporting of the actions of the simulation on a remote computer such as Clark, the user pops up the menu Window, chooses remote console and then selects which Explorer to monitor. The actions of the simulation on that computer are directed to the remote console window, in this case, clark console.

Figure 6


Applications File Edit Objects Tools Settings Run Window
Diagram 1
Source#1
Machine#1
Machine#2
Sink#1
Journal of Machine#1
80.52: Waiting for 1
81.52: calculation time 1440965
81.52: Sending (:ADD-ENTITY) to "Machine#2@Remote-Sim
81.52: Waiting for any message
83.51: Received (:ADD-ENTITY) from "Source#1@Simulati
83.51: Waiting for 1
84.51: calculation time 1432590
84.51: Sending (:ADD-ENTITY) to "Machine#2@Remote-Sim
84.51: Waiting for any message
Journal of Machine#2
80.52: clark::75.79: Received (:ADD-ENTITY) from "Mach
80.52: clark::80.52: Waiting for 1
81.52: clark::81.52: calculation time 1397714
81.52: clark::81.52: Sending (:ADD-ENTITY) to "Sink#1@
81.52: clark::81.52: Waiting for any message
83.51: clark::81.52: Received (:ADD-ENTITY) from "Mach
83.51: clark::83.51: Waiting for 1
clark Console
75.79: sending (:ADD-ENTITY) from "Machine#1@Simulation#1.lewis" to "Machine#2@Remote-Simulation#49.clark"
75.79: #<Machine#2> received (:ADD-ENTITY) from "Machine#1@Simulation#1.lewis"
80.52: Received wakeup
80.52: Notifying main sim (:REMOTE-SIMULATION-WAITING "clark" 81.52122)
81.52: Received wakeup
Console
75.79: sending (:ADD-ENTITY) from "Machine#1@Simulation#1.lewis" to "Machine#2@Remote-Simulation#49.clark"
80.52: Notifying Remote-Simulation#49.clark: (:WAKEUP 80.52122)
80.52: Received waiting until 81.52122 from "clark"
80.52: #<Source#1> sending (:ADD-ENTITY) to "Machine#1@Simulation#1.lewis"
80.52: sending (:ADD-ENTITY) from "Source#1@Simulation#1.lewis" to "Machine#1@Simulation#1.lewis"
80.52: #<Machine#1> received (:ADD-ENTITY) from "Source#1@Simulation#1.lewis"
81.52: Notifying Remote-Simulation#49.clark: (:WAKEUP 81.52122)
81.52: #<Machine#1> sending (:ADD-ENTITY) to "Machine#2@Remote-Simulation#49.clark"

To edit an object's parameters, the user double-clicks on the object. For example, double-clicking on Machine#2 will produce the choose value menu shown in Figure 7. The user can now choose what manufacturing operation this object will perform, on which Explorer to place the object, and the length of servicing necessary to do this object's processing.

The simulation can then be run, paused to change parameters and resumed using the Run menu. Upon completion of the simulation, statistics can be printed for each machine by choosing Statistics from the Window menu. The user is asked which variable to examine, for example, length-of-queue. Then, the mean, standard deviation, and minimum and maximum values, are printed for each machine (see Figure 8). Also, by double-clicking on any object with variables, such as Machine#1, the user can see the final value of the variables (also shown in Figure 8).

To give some measure of efficiency to a distributed versus a stand-alone simulation, a processing or calculation time was simulated for each machine. A loop was included as part of the code in the run method for a machine. This would mimic the amount of time a given machine might take for calculation in the real world. The loop could be adjusted based on the length of servicing time selected by a user for that machine. Total execution time for a simulation could be measured, in addition, to the number of microseconds required to complete the looping segment of the code in the run method.

Figure 7

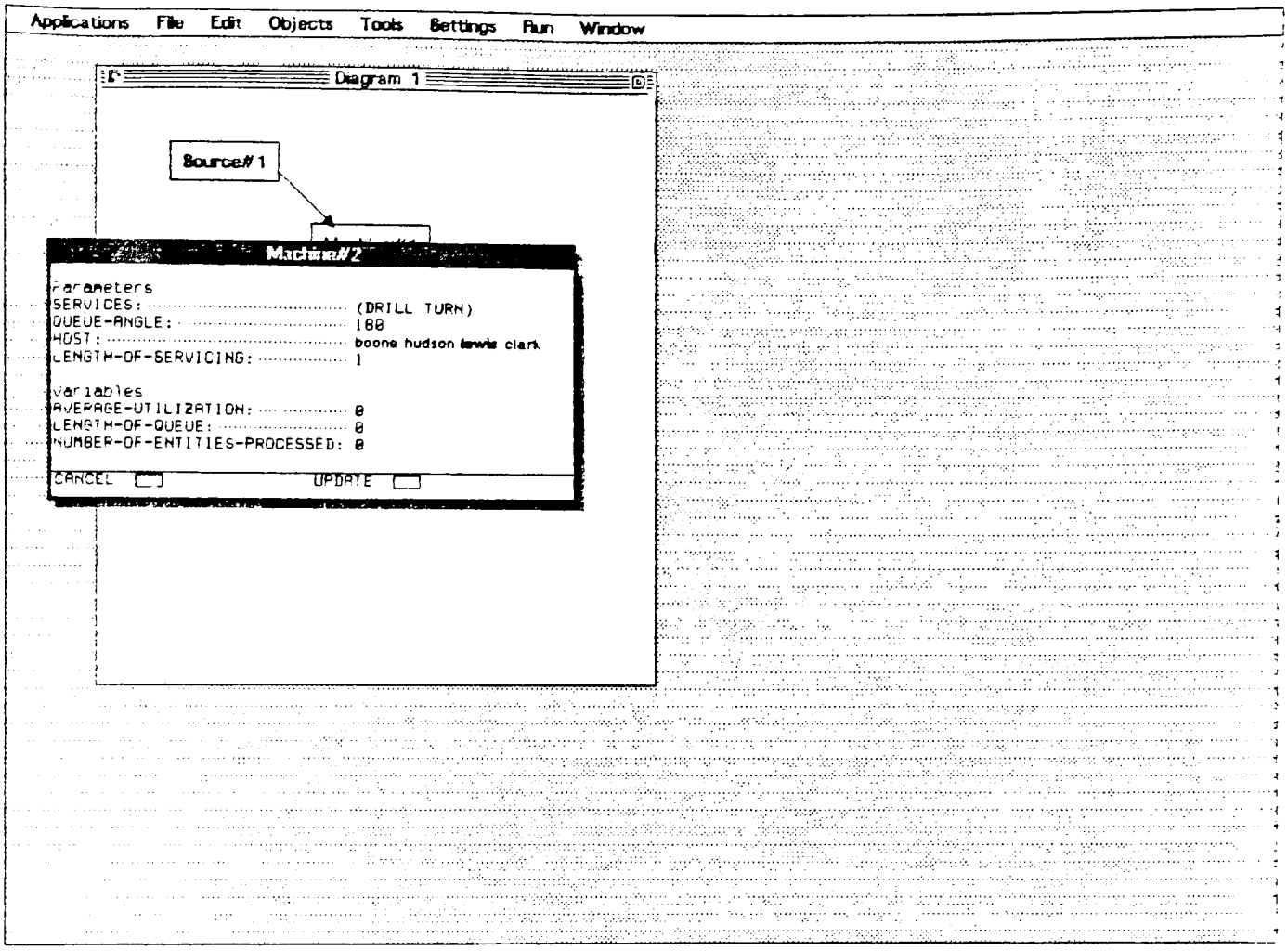
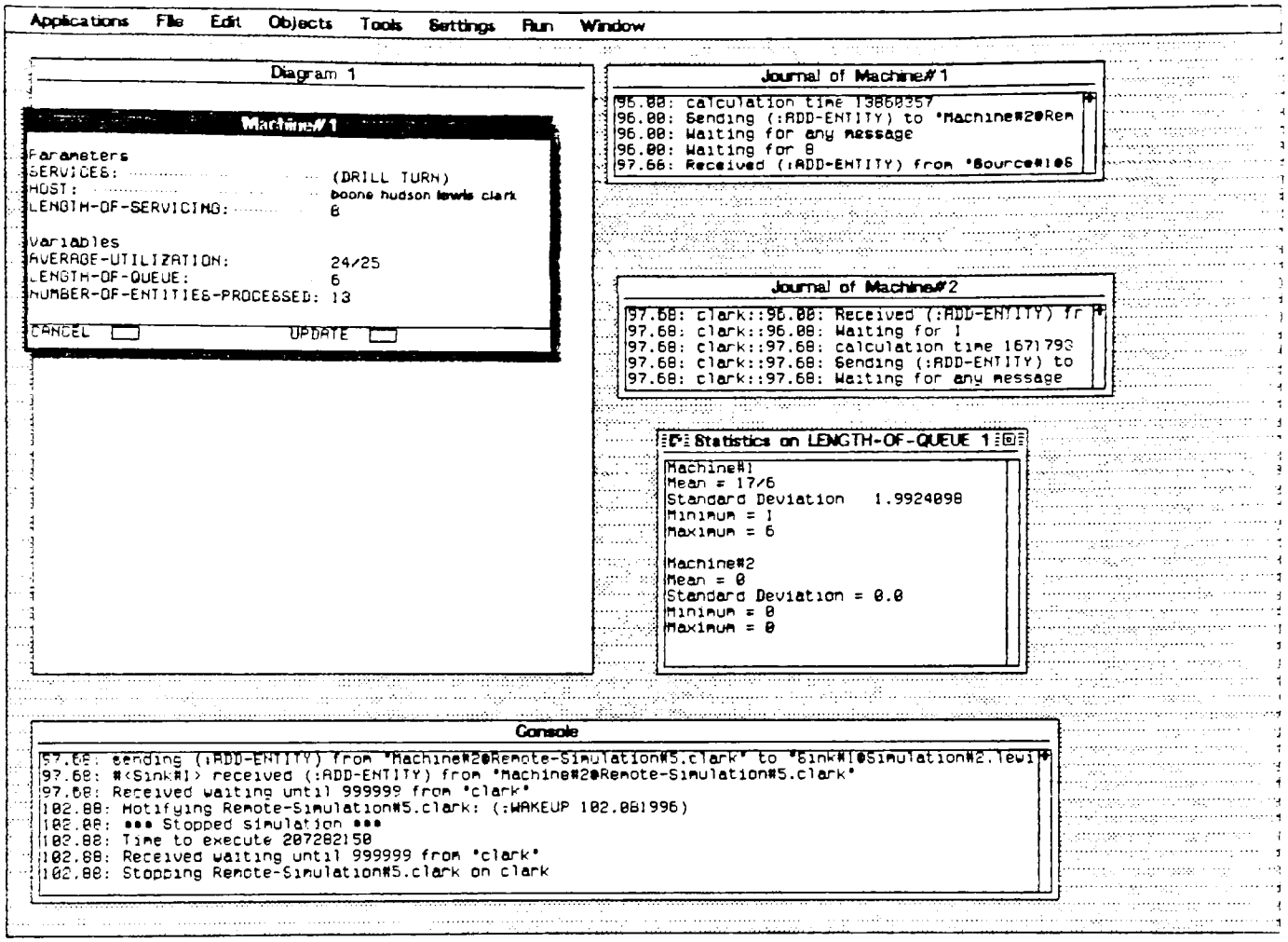

Applications
File
Edit
Objects
Tools
Settings
Run
Window
Diagram 1
Source#1
Machine#2
Parameters
SERVICES: (DRILL TURN)
QUEUE-ANGLE: 180
HOST: boone hudson lewis clark
LENGTH-OF-SERVICING: 1
Variables
AVERAGE-UTILIZATION: 0
LENGTH-OF-QUEUE: 0
NUMBER-OF-ENTITIES-PROCESSED: 0
CANCEL
UPDATE

Figure 8


Applications File Edit Objects Tools Settings Run Window
Diagram 1
Machine#1
Parameters
SERVICES: (DRILL TURN)
HOST: boone hudson lewis clark
LENGTH-OF-SERVICING: 6
Variables
AVERAGE-UTILIZATION: 24/25
LENGTH-OF-QUEUE: 6
NUMBER-OF-ENTITIES-PROCESSED: 13
CANCEL
UPDATE
Journal of Machine#1
96.00: calculation time 13860357
96.00: Sending (:ADD-ENTITY) to 'Machine#2@Rem
96.00: Waiting for any message
96.00: Waiting for 8
97.66: Received (:ADD-ENTITY) from 'Source#1@6
Journal of Machine#2
97.68: clark::96.00: Received (:ADD-ENTITY) fr
97.68: clark::96.00: Waiting for 1
97.68: clark::97.68: calculation time 1671793
97.68: clark::97.68: Sending (:ADD-ENTITY) to
97.68: clark::97.68: Waiting for any message
Statistics on LENGTH-OF-QUEUE 1
Machine#1
Mean = 17/6
Standard Deviation 1.9924098
Minimum = 1
Maximum = 6
Machine#2
Mean = 0
Standard Deviation = 0.0
Minimum = 0
Maximum = 0
Console
97.68: sending (:ADD-ENTITY) from 'Machine#2@Remote-Simulation#5.clark' to 'Sink#1@Simulation#2.lewi
97.68: #<Sink#1> received (:ADD-ENTITY) from 'Machine#2@Remote-Simulation#5.clark'
97.68: Received waiting until 999999 from 'clark'
102.88: Notifying Remote-Simulation#5.clark: (:WAKEUP 102.881996)
102.88: *** Stopped simulation ***
102.88: Time to execute 207282150
102.88: Received waiting until 999999 from 'clark'
102.88: Stopping Remote-Simulation#5.clark on clark

There are advantages and disadvantages to a distributed object-oriented system, specifically as shown by designing the widget example using the distributed simulation package on the TI Explorers.

Due to the object-oriented nature of the system, the design process was more straightforward and less time consuming, both in terms of the initial design of the manufacturing example and also for modifications and testing of different combinations of machines.

The initial design could be based on the objects already designed for the bank and car wash examples. The objects from these two examples had general characteristics which could be adapted to other situations. Also, many of the instance variables and methods could be used after some modification. Software reuse is a key advantage to object-oriented simulation. If there are a number of general purpose objects already created, building a new simulation does not have to be a time consuming or difficult process.

Once the example is built, it is easy to make changes to allow intelligent exploration of the initial example, By using pop up menus, different objects can be created graphically, placed on different computers, assigned different manufacturing tasks, and given different processing times. The simulation can then be run, paused and modifications made, and resumed. An initial problem can be easily explored by non-programmers who do not need to know how to change the code, but only have to make choices from

menus. This allows great flexibility and also rapid prototyping of different combinations of objects with varying characteristics. Use of journals to report on the execution of objects and consoles which report on the actions of the simulation objects make exploration more meaningful.

The object representation of the different components of a simulation allows for natural, concise and understandable representation of real world entities. Tellers, entrances or machines can be called by their actual names and have their characteristics described by instance variables with natural and straight-forward names such as customers, length-of-servicing and time-of-breakdown. There is a reduced semantic gap between the terminology in the real world system and the simulation.

The graphical display of the objects and their interrelationships makes it easier to use and understand the system. There is no need for the modeler to worry about complex language syntax and code.

A variety of random number generators are available to be assigned to different objects.

The efficiency of the distributed implementation is less clear-cut. The object structure makes dividing up a problem to distribute over several machines more straight-forward. However, there is a high overhead for the distributed simulation package on the Explorers and probably for most any network. An average simulation will probably take longer with this distributed system than on a single computer. However, as the amount of calculation required on each machine or object increases, the distributed system becomes

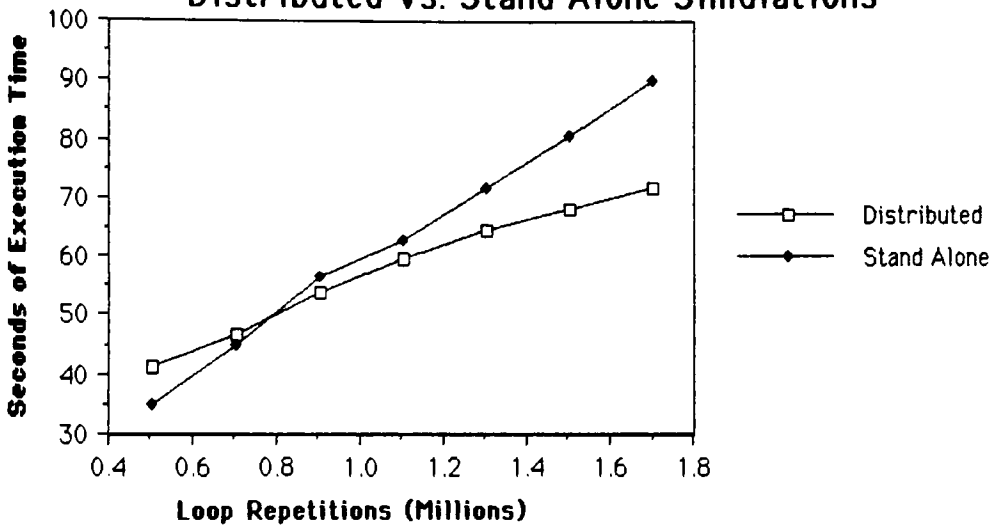
better, as the initial startup time is compensated for.

I ran the sample manufacturing program illustrated in Figure 6 using the distributed simulation package, with Machine#2 both as a remote object on Clark and locally on Lewis, another Explorer. I varied the calculation time for Machine#2 by changing the number of loop iterations in the machine run method. As can be seen by the graph in Figure 9, the simulation with the remote object started out more slowly but eventually surpassed the stand alone simulation. The remote simulation became more efficient at about 800,000 loop repetitions. This represented about 1,035,000 microseconds of calculation time on the Explorers.

Another disadvantage we discovered was the difficulty of getting such a package running. It was hard to find out the finer points of how the networking communication worked (and these were on the same type of machines, communicating among different types of computers could be even more difficult).

The Lisp environment was an advantage in getting a prototype such as this package running. Code could be easily modified. Then, instead of recompiling the whole package, only the changes needed to be recompiled and loaded. Also, it was easy to move between methods or sections of code in order to make changes. The inspector allowed close examination of the main simulation and the remote-eval function allowed a similar examination of the status of the remote simulations. Furthermore, with the TI Lisp environment, the remote network connections could remain open while changes were made. In contrast, in a Unix system, the code would need to be recompiled, reloaded,

Figure 9: Efficiency Comparison
Distributed Vs. Stand Alone Simulations



and linked, and the networking communications reestablished.

The TI environment was very tempermental and fragile. The window system frequently froze up and required rebooting and reconstructing the simulation package. Parts of the system, unrelated to the simulation package, suddenly would develop errors.

In conclusion, object-oriented design and simulation seems to be a more viable way to do simulation than the traditional programming approach. Some key advantages are the capacity for distributed simulation, the ease and capacity for intelligent exploration, code reusability, similarity of vocabulary and structure of the system in real life and in the program representation, a graphic representation of a problem and ease of modification.

Any distributed simulation would need to be chosen carefully to make sure that the computational requirements on each machine are great enough to make up for the overhead communication necessary to distribute and execute a problem.

Partly because the TI Explorers are slowly disintegrating, present plans are to port this distributed simulation package to networked Sun workstations. In planning this process, several issues need to be considered. The basic class structure is sound and should be retained, though, perhaps there is some duplication or redundancy among instance variables, for example, between remote-object-paths and remote-simulations in class simulation. Hopefully, the networking system on the Suns will be more robust. The eval-remote function will need to be rewritten based on the

communications protocol of the Sun network. The remote-connection class will need an instance variable for a Unix socket instead of the Chaos network stream. It would be a great advantage if error messages, particularly those caused by actual programming errors rather than the network, could be passed back over the network to the main simulation and not hang up the network. This would allow greater knowledge about what is actually going wrong. What sometimes appeared to be a network problem often was not.

Consideration needs to be given as to how to handle dynamic objects. Currently, they are actually messages so that the same object does not need to be recreated many times on different machines. In the original simulation package, they were actual objects and thus could inherit from other classes and easily store much more data, such as creation time. This made more statistics obtainable, such as throughput time for a given entity. Also a router object should be developed to allow a dynamic object to be directed to one of several server objects, for example, the grinding machine with the shortest queue.

Certain measures need to be seriously evaluated to improve the efficiency of the package when it is ported to the Suns. Much of what slows down the distributed simulation on the TI Explorers is the need for many processes going on, with the overhead associated with each one. If this overhead could be lessened, a resulting improvement in efficiency would result. One possibility would be to try to develop 'light-weight' processes with fewer variables. Also, network

optimization could improve the distributed package. In addition, with the initialize-remote-simulations method, each simulation is totally created each time it is run. If, instead, an evaluation could be made as to whether it was being run for the first time or not, some initialization overhead could be lowered.

Other attempts to speed up the package such as Time Warp or deadlock avoidance or recovery should be investigated. Perhaps, it would be worthwhile to categorize different simulation problems as to what speedup methods might be most appropriate. For example, in feed-forward networks which would not have cycles, such as the example in Figure 6, it would not be worthwhile to add the overhead necessary for Time Warp or deadlock avoidance or detection. However, with problems with cyclic networks or central server networks with nested cycles, deadlock could be a significant problem and would need to be addressed or else Time Warp considered as an alternative.

Another needed improvement is the ability to save diagrams and nest them within other diagrams for more complex systems and software reuse. Object and module libraries could be created. This would lead to the greater ease of programming and a capacity for scaling up to larger problems.

BIBLIOGRAPHY

Abed, S. et al. 'A Qualitative Comparison of Three Simulation Languages: GPSS/H, SLAM, Simscript'. Computers and Industrial Engineering. Vol.9 No.1 1985.

Abed, S. et al. 'A Quantitative Comparison of Three Simulation Languages: GPSS/H, SLAM, Simscript'. Computers and Industrial Engineering. Vol.9 No.1 1985.

Adelsberger, H. et al. 'Rule-based Object-oriented Simulation Systems'. Proceedings of Conference on Intelligent Simulation Environments. January 1986.

Adam, N., Dogramaci, A. Current Issues in Computer Simulation. Academic Press 1979.

Bagrodia, R., Chandy, K., Misra, J. 'A Message-Based Approach to Discrete-Event Simulation'. IEEE 13:6. June 1987.

Banks, J., Carson, J. 'Process-interaction Simulation Languages'. Simulation. May 1985.

Beckman, B., DiLoreto, M. et al. 'Distributed Simulation and Time Warp Part I: Design of Colliding Pucks'. Proceedings of SCS Conference on Distributed Simulation 1988.

Berry, O., Jefferson, D. 'Critical Path Analysis of Distributed Simulation'. Proceedings of SCS Conference on Distributed Simulation 1985.

Bezivin, J. 'Some Experiments in Object-Oriented Simulation'. Proceedings of OOPSLA 1987.

Bezivin, J. 'TimeLock: A Concurrent Simulation Technique and its Description in Smalltalk-80'. Proceedings of Winter Simulation Conference 1987.

Bezivin, J. 'An Evaluation Environment for Concurrent Object-oriented Simulation'. Proceedings of SCS Conference on Distributed Simulation 1988.

Bezivin, J. 'Design and Implementation Issues in Object-oriented Simulation'. Simuletter. June 1988.

Birtwistle, G. et al. Simula Begin. Auerback 1973.

Bobrow, D. et al. 'Common Loops Merging Lisp and Object-oriented Programming'. Proceedings of OOPSLA 1986.

Bratley, P., Fox, B., Schrage,L. A Guide to Simulation. Springer-Verlag 1987.

Calhoun, J., Lin, L. 'Support for Concurrent Simulation on Microcomputers'. SCS Conference on Modeling and Simulation on Microcomputers 1988.

Chandrasekaran, U., Sheppard, S. 'Discrete Event Distributed Simulation - A Survey'. Proceedings of Conference on Methodology and Validation 1987.

Chandy, K., Misra, J. 'Distributed Simulation: A Case Study in Design and Verification of Distributed Programs'. IEEE Transactions on Software Engineering. Vol.SE-5.No.5. September 1979.

Chandy, K., Misra, J. 'Asynchronous Distributed Simulation via a Sequence of Parallel Computations.' Communications of the ACM: Vol.24. No.11. April 1981.

Davis, D., Pegden, C. 'Introduction to Siman'. Proceedings of Winter Simulation Conference 1987.

Dutko, J. 'Using Ada for Process-interaction Oriented Discrete Event Simulation'. MS. Thesis. Rochester Institute of Technology. March 1988.

Fishman, G. Principles of Discrete Event Simulation. John Wiley and Sons 1978.

Fox, M. et al. 'Simulation Craft: An Expert System for Discrete Event Simulation'. Simulators III: Proceedings of Society of Computer Simulation Conference 1986.

Franta, W. The Process View of Simulation. North-Holland 1977.

Fujimoto, R. 'Performance Measurements of Distributed Simulation Strategies'. Proceedings of SCS Conference on Distributed Simulation 1988.

Fujimoto, R., Tsai, J., Gopalakrishnan, G. 'The Roll Back Chip: Hardware Support for Distributed Simulation Using Time Warp.' Proceedings of SCS Conference on Distributed Simulation 1988.

Gagni, A. 'Rollback Mechanisms for Optimistic Distributed Simulation Systems.' Proceedings of SCS Conference on Distributed Simulation 1988.

Gates, B. 'An Empirical Study of Time Warp Request Mechanisms'. Proceedings of SCS Conference on Distributed Simulation 1988.

Gilmer, J. 'An Assessment of Time Warp Parallel Discrete Event Simulation Algorithm Performance'. Proceedings of SCS Conference on Distributed Simulation 1988.

Goldberg, A., Robson, D. Smalltalk-80 The Language and Its Implementation. Addison-Wesley 1983.

Grain, R. et al. 'Advanced Features of GPSS/H'. Proceedings of Winter Simulation Conference 1987.

Haider, S., Banks, J. 'Simulation Software Products for Analyzing Manufacturing Systems'. Industrial Engineering 18:7 1986.

Jefferson, D. 'Fast Concurrent Simulation Using the Time Warp Mechanism.' Proceedings of SCS Conference on Distributed Simulation 1985.

Jefferson, D. 'Implementation of Time Warp on the Caltech Hypercube'. Proceedings of SCS Conference on Distributed Simulation 1985.

Jefferson, D. 'Virtual Time'. ACM Transactions on Programming Languages and Systems'. Vol.7. No.3 .July 1985.

Kaudel, F. 'A Literature Survey on Distributed Discrete Event Simulation.' Simuletter. Vol.18 No.2. June 1987.

Kerr, R., Percival, D. 'Use of Object-oriented Programming in a Time Series Analysis System'. Proceedings of OOPSLA 1987.

Kim, T., Ziegler, B. 'The Class Kernel-Models in DEVS-Scheme: A Hypercube Architecture Example'. Simuletter. June 1988.

Klahr, P. 'Expressibility in Ross: An Object-oriented Simulation System'. AI Applied to Simulation: Proceedings of the European Conference at the University of Ghent 1985.

Klahr, P., et al. 'TWIRL: Tactical Warfare in the Ross Language'. Expert Systems: Techniques, Tools and Application. Addison-Wesley 1986.

Knapp, V. 'The Smalltalk Simulation Environment'. Proceedings of the Winter Simulation Conference 1986.

Knapp, V. 'The Smalltalk Simulation Environment, Part II'. Proceedings of the Winter Simulation Conference 1987.

Li, X., Unger, B. 'Languages for Distributed Simulation'. Proceedings of Conference on AI and Simulation 1987.

Lomow, G., Cleary, J., et al. 'A Performance Study of Time Warp'. Proceedings of SCS Conference on Distributed Simulation 1988.

MacArthur, D., Klahr, P., Narain, S. 'Ross: An Object-oriented Language for Constructing Simulations'. Expert Systems: Techniques, Tools and Applications. Addison-Wesley 1986.

MacIntosh, D., Conry, S. 'SIMULACT: A Generic Tool for Simulating Distributed Systems'. Northeast Artificial Intelligence Consortium. 1987.

Madisetti, V., Wairand, J., Messerschmitt, D. 'WOLF: A Roll-back Algorithm for Optimistic Simulation Systems'. Proceedings of Winter Simulation Conference 1988.

Marti, J. 'RISE: The Rand Integrated Simulation Environment'. Proceedings of SCS Conference on Distributed Simulation 1988.

McFall, M., Klahr, P. 'Simulation with Rules and Objects'. Proceedings of the Winter Simulation Conference 1986.

Middleton, S., Zanconato, R. 'Blobs : An Object-oriented Language for Simulation and Reasoning'. AI Applied to Simulation: Proceedings of the European Conference at the University of Ghent 1985.

Misra, J. 'Distributed Discrete-Event Simulation'. Computing Surveys Vol.18 No.1. March 1986.

Moon, D. 'Object-oriented Programming with Flavors'. Proceedings of OOPSLA 1986.

Nicol, D. 'High Performance Parallelized Discrete Event Simulation of Stochastic Queuing Networks'. Proceedings of Winter Simulation Conference 1988.

Nielsen, N. 'Applicability of AI Techniques to Simulation Models'. Simulators IV: Proceedings of Society for Computer Simulation 1987.

O'Reilly, J., Lilegdon, W. 'SLAM II Tutorial'. Proceedings of Winter Simulation Conference 1987.

Pascoe, G. 'Encapsulators: A New Software Paradigm in Smalltalk-80'. Proceedings of OOPSLA 1986.

Pazirandeh, M., Becker, J. 'Object-oriented Performance Models with Knowledge-based Diagnostics'. Proceedings of Winter Simulation Conference 1987.

Peacock, J.K., Wong, J.W. and E. Manning. 'Distributed Simulation Using a Network of Processors'. Computer Networks, 3, North Holland Publishing Company 1979.

Pliske, D. 'Computer System Simulation in Scheme.' Simulators IV: Proceedings of Society for Computer Simulation Conference 1987.

Power, L., Weiss, A., ed. 'Addendum to the Proceedings'. Proceedings of OOPSLA 1987.

Quinlivan, B. 'Deadlock Resolution in Distributed Simulation'. Master's Thesis, Computer Science Department, University of Texas at Austin 1979.

Reed, D., Malony, A., McCredie, B. 'Parallel Discrete Event Simulation Using Shared Memory.' IEEE Transactions on Software Engineering. Vol.14 No.4. April 1988.

Reed, D., Malony, A. 'Parallel Discrete Event Simulation: The Chandy-Misra Approach'. Proceedings of SCS MultiConference on Distributed Simulation 1988.

Reynolds, P., Kuhn, C. 'A Performance Study of Three Protocols for Distributed Simulation'. Proceedings of Conference on Methodology and Validation 1987.

Roberts, S., Heim, J. 'A Perspective on Object-Oriented Simulation'. Proceedings of Winter Simulation Conference 1988.

Rothenberg, J. 'Object-oriented Simulation: Where Do We Go From Here'. Proceedings of Winter Simulation Conference 1986.

Rothenberg, J. 'Knowledge-based Simulation at Rand'. Simuletter. June 1988.

Russell, E. 'Simscript II.5 and SimAnimation'. Proceedings of Winter Simulation Conference 1987.

Schriber, T. 'Perspectives on Simulation Using GPSS'. Proceedings of Winter Simulation Conference 1987.

Seethalakshmi, M. 'A Study and Analysis of Performance of Distributed Simulation'. Master's Thesis, Computer Science Department, University of Texas at Austin 1979.

Stairmand, M, Kreutzer, W. 'POSE: A Process Oriented Simulation Environment Embedded in Scheme'. Simulation. April 1988.

Standridge, C et al. 'A Tutorial on TESS: The Extended Simulation Support System'. Proceedings of Winter Simulation Conference 1987.

Snyder, A. 'Encapsulation and Inheritance in Object-oriented Programming Languages'. Proceedings of OOPSLA 1986.

Tello, E. 'Object-oriented Programming'. Dr. Dobb's Journal. March 1987.

Tello, E. 'Object-oriented Programming in AI'. Dr. Dobb's Journal. April 1987.

Texas Instruments Explorer Manual. Flavors (Chapter 2).

Ulgen, O., Thomasma, T. 'Simulation Modeling in an Object-oriented Environment Using Smalltalk-80'. Proceedings of Winter Simulation Conference 1986.

Unger, B., Lomow, G., Birtwistle, G. Simulation Software and Ada. Society for Computer Simulation 1984.

Unger, B., Trybul, T., Lomow, G. Simulation in Ada. Society for Computer Simulation 1985.

Unger, B., Lomow, G., Andrews,K. 'A Process Oriented Distributed Simulation Package'. Proceedings of SCS Conference on Distributed Simulation 1985.

Unger, B.W. 'Object-oriented Simulation - Ada, C++, Simula'. Proceedings of Winter Simulation Conference 1986.

Weiner, R. 'Object-oriented Programming in C++ A Case Study' SIGPLAN Notices Vol.22 No.6. June 1987.

Weiner, R., Pinson, L. An Introduction to Object-oriented Programming and C++. Addison Wesley 1988.

APPENDIX

ART-ROSS: an integrated rule and object based implementation and extension of the Ross language; conversion from procedural nature of Ross to rule based implementation in ART
reference - Philip Klahr of the Inference Corp, 1986

AutoMod/AutoGram: limited simulation application package for manufacturing and material handling systems; preprocessor to GPSS/H; AutoGram provides animation for AutoMod
reference - AutoSimulations, Inc., Bountiful, UT 1986

BLOBS: Blackboard Objects; object-oriented language which provides a framework for design and building of both simulation and reasoning systems involving multiple real time objects; domain of monitoring aircraft tracks on radar
reference - Michael Bell for Royal Signals and Radar Establishment, Great Britain

CommonObjects: an object-oriented extension to Common Lisp; allows designer to specify type hierarchy independently of inheritance hierarchy; restricts access to inherited instances to allow more flexible remodeling
reference - Alan Snyder, 1985

Demos: simulation language based on Simula
reference - Dahl, 1966

DEVS-Scheme: realization of DEVS (Discrete Event System Specification) formalism in a Lisp-based object-oriented environment; coded in SCOOPS, an object-oriented superset of PC Scheme; supports hierarchical modular specification of discrete event models
reference - Ziegler, 1986

Director: an object-oriented language based on the actor paradigm
reference - K. Kahn, MIT, 1979

GASP II: Fortran based simulation package
reference - Pritsker and Kiviat, 1969

HOOD: Hierarchical Object Oriented Design; has active and passive object concepts
reference - Cri, 1987

Loops: experimental knowledge programming system which integrates object-oriented programming, procedure-oriented programming, access-oriented programming and rule-oriented programming
reference - Stefik, 1983

KBSim: Knowledge-Based Simulation project at Rand Corp; combine simulation, reasoning and graphics in order to make simulation more powerful and comprehensible
reference - Rand Corp.

MAP/1: simulation based Modeling and Analysis Program (MAP) used to design and evaluate batch manufacturing systems; used to analyze job shops, flow shops, assembly lines, computerized manufacturing systems;
reference - Pritsker and Associates, Inc.

MicroNet: process flow simulation system for microcomputers; can operate in an interrupt mode so that interim statistical reports can be prepared, specific network values queried or runtime parameters changed
reference - Pritsker and Associates, Inc.

ObjectLisp: an object-oriented extension to Common Lisp; eliminates any special syntax for message passing; object-oriented methods invoked with same system as any Common Lisp function
reference - G.Dresher, J.Ressler, 1985

Pascal Plus: special language for simulation from Pascal and Simula
'doesn't offer enough new features to justify wide acceptance', Bratley, 1987

PASSIM: set of procedures that can be used to construct discrete event simulation models in Pascal; combines scheduler and entity concepts from GPSS with pointer based data structures and control structures from Pascal
reference - C.C.Barnett, Walla Walla College

Plasma: object-oriented language
reference - C.Hewitt, 1977

RESQ: discrete general purpose process-oriented language with a choice of solution techniques for a network of queues
reference - E.MacNair, IBM

Samoa: discrete event simulation package built upon Ada; based on Demos and Simula;
reference - G.Lomov, B.Unger, 1982

SEE-WHY/Witness: limited simulation application package for manufacturing and material handling systems
reference - T.Lee Simulation Vol.45 No.2 1985

Siman: Simulation Analysis; general purpose program for modeling combined discrete/continuous systems; animation provided by Cinema
reference - C.Pegden, 1982

SIMKIT: system developed by IntelliCorp for building discrete event simulation; based on KEE; commercially available; capabilities include basic modeling facilities, more general knowledge-based reasoning techniques, object-oriented programming; frame structures for representing objects; graphical representation and manipulation of entities in model;
reference - IntelliCorp, 1985

Simone: client server style of programming using Hoare monitor mechanisms
reference - Kaubisch, 1976

Simulation Craft: system developed by Carnegie Group to assist in design and use of simulation model as well as to assist in the building of that model; commercially available
reference - N.Sathi, 1986

SWIRL: simulating strategic warfare in the Ross simulation language
reference - Rand Corp, 1982

TESS: enables graphical construction of a SLAM network model; translation of graphical input to statement form; database management; output analysis; scenario animation
reference - Pritsker Associates

ThingLab: constraint oriented simulation lab
reference - Alan Borning, 1981

Trellis/Owl: an object-oriented language from DEC; semantics model inheritance graph directly; restricts access to inherited instance variables for more flexible remodeling
reference - Craig Schaffert, 1986

TWIRL: simulating tactical warfare in the Ross simulation language
reference - Rand Corp, 1984