AN ABSTRACT OF THE THESIS OF

Donald R. McGregor for the degree of Master of Science in Industrial and Manufacturing Engineering presented on January 14, 1992.

Title: An Interactive Object-Oriented System for Discrete Simulation Modeling and Analysis

Abstract approved: __________________________

Sabah Randhawa

One of the commonly used simulation approaches is process orientation. This is based on the use of nodes (or blocks) that perform functions in series. In spite of the compactness and ease of learning that characterize process-based simulation, many languages are somewhat complex, primarily the result of the large number of nodes that users have to deal with and the considerable gulf between a user's abstract notion of the model and the details required to implement it.

This paper describes a process-based simulation system that integrates object-oriented programming, visual interactive simulation and graphical model specification. Object-oriented programming techniques and simulation seem to be a natural match. The process classes are represented as network blocks or network nodes, and the process as a network diagram or directed graph. Arcs connect the nodes and specify the next step in the process. Each block type has its own icon. Developing an application model requires selecting a set of nodes, connecting them, and specifying the parameters (such as activity durations and random number streams) of the nodes through dialog boxes or inspection panels. Nodes have been designed to accomplish the major requirements in simulation modeling, including creation and termination of entities, attribute assignment, branching, queues and resources, activity specification and statistics collection and display. Additional system features include: statistics manipulation for steady state results, execution trace utilities, and limited animation capabilities.

The system has been implemented for the NeXT programming environment using Objective-C. The NeXT includes an extensive object-oriented user interface library, relatively powerful hardware, and a modern multi-tasking and virtual
memory operating system. Objective-C allows object-oriented concepts such as inheritance and subclassing while adding only a few constructs to that of the C language.

The system modeling environment developed in this research enhances the applicability and usability of high level modeling tools. The program also provides a platform for further work on the distribution of the modeling process over several cooperating, communicating applications.
Copyright Notice

Computer source code and NeXT Interface Builder files Copyright 1991, 1992 Donald R. McGregor. All rights reserved. The source code may not be used in another project without the prior written permission of the author.
An Interactive Object-Oriented System for Discrete Simulation Modeling and Analysis

by

Donald R. McGregor

A THESIS submitted to Oregon State University

In partial fulfillment of the requirements for the degree of Master of Science

Completed: January 14, 1992
Commencement: June 1992
APPROVED:

Redacted for Privacy

Dr. Sabah Randhawa, Associate Professor, Industrial and Manufacturing Engineering in charge of major

Redacted for Privacy

Dr. Thomas West, Chairman, Department of Industrial and Manufacturing Engineering

Redacted for Privacy

Dean of Graduate School

Date Thesis is Presented: January 14, 1992
Typed by Donald R. McGregor for: Donald R. McGregor
Acknowledgement

Thanks to my parents, to the OSU faculty members that supplied invaluable advice, support, and motivation, and to the men and women of Superette market on Monroe street for keeping the Big Gulp machines running.
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The opening screen</td>
<td>21</td>
</tr>
<tr>
<td>2. A network block</td>
<td>22</td>
</tr>
<tr>
<td>3. The create node</td>
<td>23</td>
</tr>
<tr>
<td>4. Create node inspector panel</td>
<td>24</td>
</tr>
<tr>
<td>5. The destroy node</td>
<td>26</td>
</tr>
<tr>
<td>6. Destroy node inspector panel</td>
<td>27</td>
</tr>
<tr>
<td>7. The assign node</td>
<td>28</td>
</tr>
<tr>
<td>8. Assign node inspector panel</td>
<td>30</td>
</tr>
<tr>
<td>9. The activity node</td>
<td>32</td>
</tr>
<tr>
<td>10. Activity node inspector panel</td>
<td>32</td>
</tr>
<tr>
<td>11. The branch node</td>
<td>34</td>
</tr>
<tr>
<td>12. Branch node inspector panel</td>
<td>34</td>
</tr>
<tr>
<td>13. The arithmetic node</td>
<td>36</td>
</tr>
<tr>
<td>14. Arithmetic node inspector panel</td>
<td>36</td>
</tr>
<tr>
<td>15. The statistics node</td>
<td>37</td>
</tr>
<tr>
<td>16. Statistics node inspector panel</td>
<td>38</td>
</tr>
<tr>
<td>17. The queue node</td>
<td>39</td>
</tr>
<tr>
<td>18. Queue node inspector panel</td>
<td>40</td>
</tr>
<tr>
<td>19. The main menu</td>
<td>44</td>
</tr>
<tr>
<td>20. The Simulation submenu</td>
<td>45</td>
</tr>
<tr>
<td>21. TV Inspection and repair line schematic diagram</td>
<td>49</td>
</tr>
<tr>
<td>22. Initial node layout for TV inspection</td>
<td>50</td>
</tr>
<tr>
<td>23. The connected TV inspection model</td>
<td>51</td>
</tr>
<tr>
<td>24. TV inspection queue panel</td>
<td>52</td>
</tr>
<tr>
<td>25. Schematic diagram of quarry operations</td>
<td>55</td>
</tr>
<tr>
<td>26. Initializing truck entities</td>
<td>57</td>
</tr>
<tr>
<td>27. Quarry operations model</td>
<td>58</td>
</tr>
<tr>
<td>28. Tracing program execution</td>
<td>59</td>
</tr>
<tr>
<td>29. Return trip inspector panel</td>
<td>60</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Random number generator parameters</td>
<td>18</td>
</tr>
<tr>
<td>2. Quarry operation times</td>
<td>55</td>
</tr>
</tbody>
</table>
AN INTERACTIVE OBJECT-ORIENTED SYSTEM FOR DISCRETE SIMULATION MODELING AND ANALYSIS

I. Simulation Modeling

I.1. The need for simulation.

Former Xerox PARC employee and Apple Fellow Alan Kay has said that models are the modern scientist's voodoo dolls. We create abstractions of real systems, then experiment with the model—pushing in pins here, changing this or that feature, modifying some input. What we observe about the response of the model tells us something, we hope, about the real system.

For centuries we have created our voodoo dolls using mathematics and achieved excellent results. We have no problems exactly predicting planetary orbits or electromagnetic phenomena with the aid of elegant and precise mathematical and analytical models. However, by necessity the models we build of most systems are simpler and in some ways cruder than the systems they claim to represent; any real system is enormously complex and almost impossible to exactly reproduce, even if that was what we wanted to do. Instead we model only the aspects of the system that interest us and simplify, abstract, or ignore things that don't.

But complex (and sometimes not-so-complex) systems are difficult to describe using mathematics, either because the essence of the system cannot be captured in equations or because the solution to the model becomes intractable. We might make simplifying assumptions about the system, but too many inappropriate assumptions will reduce the model's correspondence to the real system. The scientist is caught in a choice between a realistic but unsolvable model and a model that can be solved but bears little resemblance to the problem at hand.

It is for these types of problems that simulation is best suited. The system can be mimicked to an arbitrary level of precision, and then its response to the experimentalist's pin-pricks and manipulations observed. The simulation model does not yield a closed form solution as does the analytical model, so to be confident of the model's results we need to poke and prod and then repeat our
experiments under a variety of conditions. But they can give insight and quantitative results when analytical approaches are impossible.

Simulation has been used as an explanatory device to define a system; as an aid to determine critical system elements; as an assessment tool for alternative designs; and to predict future system states (Pritsker and Pegden, 1979). Its application to a wide range of industry, government, and research problems has proven its worth many times over.

1.2. Use of simulation languages.

The practice of simulation is intimately intertwined with the digital computer. From the outset of the computer age simulation has been an important application, and on occasion it has driven further hardware and software developments in the computer industry. The practice of simulation model building has paralleled that of the rest of the software industry—from machine level programming with switches or assembler, to high level languages such as FORTRAN or C, to specialized languages and graphical model specification tools. Each step has reduced the effort required to perform simulation experiments, occasionally at the cost of being able to do exactly as one pleased.

1.3. Some trends in the field.

As new software and more powerful computers become available the techniques and tools of simulation have evolved.

1.3.a. Integration of the simulationist’s environment. In the late 70’s and early 80’s some thought that an integrated environment for simulation that combined model building, execution, result analysis, and presentation in one package would ease the simulation analyst’s job. Standridge writes (Standridge, 1985):

Integration hides many of the technical details required to perform a simulation project but not directly related to problem solving. Flexible data selection for reports, graphs, and animations can be provided with the mechanics of the data selection hidden from the user. In addition integration allows the user to learn a single software package to perform an entire simulation project.
Standridge was correct in observing that the user should not need to know the intricate details of transferring data from one independent module to another during the simulation process, and correct in noting that greater integration of the steps in an analyst's tasks would enable more attention to be paid to the problem at hand rather than the mechanics of transferring information among several stand-alone programs. However, the integration he wrote about seems to be occurring at a higher level than that of a single program. The operating environment is the level at which the software industry is standardizing.

Current software practice has, in general, moved away from a "vertical integration" orientation in which all operations a user might need to perform a specific task are included in a single software package, and is moving towards a "horizontal integration" model, in which programs and utilities perform only one operation in a user's task. Data is created in one program, and then sent to another for specialized processing and analysis. Each program does its part of the task well and relies on other applications to perform the remaining operations, and all the programs have a similar "look and feel" that reduce the user's learning time.

It is becoming more common, for example, for programs to rely on special purpose graphing utilities instead of implementing their own plotting routines. A word processing program is meant to be a writing tool; any imbedded graphics creation facilities are not likely to be as rich and powerful as those found in a program whose only purpose is graphics. Similarly, a dedicated statistics program is likely to offer more data analysis features than those included in a program that must also do simulation and graphing.

The Macintosh, Windows, and some windowing Unix implementations can easily and transparently (and in some cases automatically) transfer data among applications. Programs written for a specific environment have similar user interfaces, which reduce the effective distinctions among programs—it becomes difficult to know when you've left one program and entered another since all the programs within an operating environment operate in essentially the same way. The operating environment specifies the rules for exchanging data among programs, and provides programming constructs that make a developer's compliance with the rules less painful than creating a new and incompatible method. Software
companies have strong incentives to comply with the rules to ensure their product has the widest acceptance possible.

Under these conditions the user may begin to put together a suite of applications that perform all the tasks of an integrated program, and transparently use them in his analysis without the problems Standridge points out. Each application is likely to do its job better than its corresponding function in an integrated package, since it concentrates on doing only one thing well. All the programs have a similar user interface that conform to the standards specified for the operating environment, so users do not become disoriented when jumping between applications. Individual elements of the program suite can be replaced when more useful applications from other vendors enter the market.

The analyst’s environment is integrated—but the integration is among applications and within the operating environment, not within a single monolithic application.

I.3.b. Graphical Specification of Models and Visual Interactive Simulation. Corresponding trends, and ones well documented in the literature, are the specification of simulation models through graphical techniques, also called “no-programming model building”, and Visual Interactive Simulation (Bell and O'Keefe, 1987; Hurrion, 1980; Mathewson, 1984).

Conventional simulations are completely specified before its execution begins and then proceed to their conclusion without any input from the user. Animation capabilities that show an iconic representation of the system as the model runs are often used to display the model’s status during execution of the program. Visual Interactive Simulation (VIS) takes a somewhat different approach. While it also displays the system status to the user through animation or other graphics, it allows the user to interact with a running simulation and change its parameters or course. The user is no longer a spectator but rather an active participant—changing the path of execution when unusual circumstances arise, modifying the parameters of the model, and stopping or starting it to examine its state.
A number of benefits are ascribed to VIS, including user satisfaction, the ability of the user to shift attention among different parts of the simulation, user recognition of unusual system states, and increased confidence in the model's validity (Bell and O'Keefe, 1987). However, there are dangers with the technique, as pointed out in the same Bell and O'Keefe article and discussed in the 1990 Winter Simulation Conference panel on interactive simulation (Matwicazk, 1990; Davis, 1990; Musselman, 1990; Brunner, 1990), including the statistical validity of the performance parameters collected and a tendency to perform analysis by just looking at the screen—a screen that might well reflect transient or unrepresentative states—rather than by using rigorous statistical methodologies.

Norman (1988) states that the designer of "everyday things" should create an understandable conceptual model and make it visible to the user. While discrete event simulation tools are not "everyday things" in the sense that VCRs or cars are, the same care that goes into designing good mass market consumer goods should be put into other human endeavors.

Norman (1988) observes that people are able to use a great many objects they have never seen before, because "...not all of the knowledge required for precise behavior has to be in the head. It can be distributed—partly in the head, partly in the world, and partly in the constraints of the world." Exacting detail is not always required, natural constraints are present, and cultural restrictions circumscribe the range of actions that are meaningful in many of our everyday interactions with objects (Norman, 1988). The user need not, and should not, be required to internalize many fundamentally arbitrary facts and rules to go about his business, though a minimum are generally required for complex systems. Norman calls ideas communicated to the user through the object's immediately apparent characteristics the system image—the cues in the object and the environment that tell the user about how to use objects, ideally without reference to manuals or instructions.

Apple Computer suggests a number of guidelines for graphical user interface programmers that help achieve a good system image. These include metaphors from the real world; direct manipulation; see-and-point (instead of remember-and-type); consistency; what you see is what you get; user control;
feedback and dialog; forgiveness; perceived stability; aesthetic integrity; solid
graphical communication principles; visual consistency; simplicity; and clarity
(Apple Computer, 1987).

Whether explicitly recognized or not, efforts to create a clear and
comprehensible system image underlay another trend in simulation tools: the
graphical specification of models. More immediately apparent information about
how to use some feature of a program can be encoded into a visible object than are
likely to be incorporated into, say, the text editor for the programming-based model
specification tool. The user is not required to memorize as many arbitrary facts—
the order in which function arguments are passed, or the spelling of a statement—
and more cues can be given, such as the random number distributions that are
available at a certain point in the specification of a model. Even non-programmers
can create simulations. These gains are sometimes balanced by a loss in the
flexibility that can be provided by an expressive programming language
(Mathewson, 1984), just as some flexibility was lost in the move from general-
purpose programming languages to simulation languages.

Norman describes two ways of getting a task done—by issuing commands
to someone else who does the actual work, or by doing the operations yourself. The
first method is effective when the task is laborious and repetitive, the second when
the task is critical, novel, or ill-defined (Norman, 1988). These modes also exist in
computer systems. Most systems have command-mode, third person interaction:
"Give me a directory listing," or another command from a set of possible verbs.
Others have a direct manipulation interface, in which the user is made to feel as if
he is in control of the system himself, without a mediating third party.

Simulation modeling can fall into either category. Building a model is an
exercise in the creative abstraction of critical features of the real-world system,
fraught with trial and error, and full of unseen pitfalls and logical traps. Once the
model has been built it may be exercised with different input states and repeatedly
run to test the sensitivity of the results and to perform statistical analysis. VIS
seems to help immensely with model building and verification, while being less
valuable or even a hindrance when conducting repeated runs with new data.

---

1This problem is also expressed as the "separation of the model from the experiment."
A number of programs have attempted to address graphical model building, Visual Interactive Simulation, or both, including SLAMSystem (O'Reilly and Nordlund, 1989); Extend (Valdés, 1989); and research efforts including Tiny Tim (Kreutzer, 1990) and IMDE (Ozden, 1991).

I.3.c. Object Orientation. Object-oriented programming (OOP) techniques and simulation seem to be a natural match. From the original Simula language to more recent investigations with C++, Object Pascal, Objective C, or Smalltalk (Eldredge, J. McGregor, and Summers, 1990; Sanderson and Rose, 1990; Sanderson, 1990; Ozden, 199; Kreutzer, 1986, 1990; Adiga and Glassey, 1989), object orientation has been an important part of simulation methodology.

A survey of the benefits of object orientation in manufacturing systems can be found in Adiga (1989). Among the benefits cited are transferring a real world problem to a computer representation with less effort, the ability to have multiple levels of abstraction, incremental development styles, and software reuse. Cox (1990) in particular champions the reuse of software objects through the creation of "software ICs," drawing an analogy between proposed software engineering techniques and longstanding hardware development styles. Rather than designing each new circuit board from scratch, electronics engineers typically use well-tested and documented integrated circuits (ICs) and combine them in novel ways. Since the hardware designers stand on the shoulders of the chip designers, they are able to operate at a higher conceptual level and build cheap, reliable, and functional products very quickly. This stands in contrast to the near-legendary tendency for software to be late, expensive, and unreliable.

OOP is also widely used in graphical user interface development. In a project that makes use of graphical and visual displays this is further justification for using OOP.

I.4. Focus of Current Effort

I.4.a. Why the current effort? The NeXT™ computer is a fascinating system from a software development standpoint. It is object-oriented from the ground up, including an extensive object-oriented user interface toolkit with

---

2NeXT is a registered trademark of NeXT Computer, Inc.
graphical interface building tools. The NeXT uses the Mach operating system with Unix™3 extensions, providing the programmer with a large, flat, protected memory address space, preemptive multitasking, interprocess communication, virtual memory, and "lightweight processes" or threads. The system uses a single imaging model, Display PostScript™4, which allows the same code to be used for both printing graphics and displaying them on the screen. Mature and powerful development tools are to be had for free, many of them bundled with the computer or available from electronic bulletin boards and Internet archive sites. The hardware is powerful enough to support a rich programming and operating environment, and a large, high resolution screen is standard.

All these features are useful in an investigation of the concepts discussed above, including graphical, interactive, object-oriented and intercommunicating programs. The fact that the NeXT is moderately quick and that simulation makes heavy demands on the computation capability of computers are additional selling points.

Furthermore, as of this writing the NeXT has not had any simulation programs written to specifically take advantage of its user interface, NeXT Step. This leaves a serious gap for the simulation analyst who might prefer to use the NeXT.

I.4.b. Scope of the project. In an ideal world the result of this effort would be a commercial quality, easy to use, flexible, graphic and interactive simulation tool capable of describing any system. Unfortunately, all projects need to be limited, this one not excepted. To define a reasonable amount of work that could be accomplished in the space of a few months, the scope of the project was limited in the following ways:

I.4.b.i. The simulation tool would be limited to discrete-event simulation. While it is possible to develop a continuous or mixed continuous-discrete simulation tool, and these features are indispensable in some situations, they do not seem to add enough functionality for the typical user targeted by this program to offset the extra programming effort. Furthermore, the subjects being investigated—

---

3Unix is a registered trademark of Unix System Laboratories.
4PostScript is a registered trademark of Adobe Systems Incorporated.
OOP, VIS and graphical model specification, and simulation program communication with other programs—do not require that both discrete and continuous simulation capabilities be present. One or the other is adequate. Since Industrial Engineering makes more use of discrete models, this was the world view selected.

I.4.b.ii. Interprocess communication (IPC) exists only as a demonstration feature, and is not fully integrated into the program. It is not exceptionally difficult to set up cooperating programs on the NeXT. What NeXT terms Services and Speaker/Listener facilities make IPC available to any program that chooses to ask for them, and presents them to the user through the main menu automatically. However, time constraints precluded a full implementation of this feature.

I.4.b.iii. Development would concentrate on a few process nodes capable of describing many, but not all, systems. SLAM has some 22 or more network objects; SIMAN has 39 block statements. Both environments are flexible and powerful simulation tools that can describe a great many systems, but recreation of all their functionality is far too ambitious for this project. The number of network nodes was limited to eight carefully chosen, well defined node types that can describe many typical systems.

I.4.b.iv. Priority would be given to an easy-to-use interface, compliant with NeXT Step standards, that prevent the user from taking obviously senseless actions. This is simply good programming practice, but a practice that often gets lost in the competition between more features—something every technically inclined user wants—and a clear, understandable and forgiving interface that looks and behaves like other programs in the environment—which everyone wants, in principle.

I.4.b.v. The program should provide a basis for further development and experimentation. It should be expandable and provide a framework for more complete investigation of certain trends in the field.

An effort was undertaken to write such a program. The result, an application called Ents, is the primary subject of this thesis. "Ents" doesn't stand for anything; it's named for beings introduced in J.R.R. Tolkien's Lord of the Rings. The Ents carefully weighed several courses of action, then relentlessly implemented the best one.
II. Simulation Concepts

II.1. Basic requirements for simulation modeling.

II.1.a. Discrete event and process oriented simulation. A discrete event simulation advances the system clock asynchronously, from scheduled event to scheduled event. Nothing of consequence is assumed to occur between events. Process oriented simulation describes the steps an entity goes through while in the simulation— it might be created, wait in a line, undergo an activity, and then exit the system.

The two are often intimately connected. A typical approach is to combine several discrete events into a single process class, such as a queue. The entity arrives and requests a limited resource. If it is available, the entity immediately begins service; if not, the entity waits in line for the resource to be freed. Entities in service eventually finish and release the resource for the next entity, then proceed to the next process.

The process classes can be represented as network blocks or network nodes, and the process as a network diagram or directed graph. Arcs or edges connect the nodes, also called vertices. The arcs have a direction which specifies the next step in the process.

The underlying mechanism for the simulation engine is still based on the scheduling and processing of events, and the state of the system is still assumed to be unchanging between events. However, the model logic is less fragmented than in a pure discrete event approach, and the causal relationships between sequences of events are shown more clearly (Kreutzer, 1986).

II.1.b. Representing entities and attributes. Entities are objects which undergo a process. For example, a bank teller window might be simulated with a queue. Entities, representing customers, are created, enter the queue, are serviced, and then exit the system.
Attributes are qualities of individual entities. In the bank teller example, each entity might have a different name, representing the name of each customer. Attributes might also be present to represent the type of transaction the customer wants to perform, which affect how long it takes to be serviced. More than one attribute might be associated with an entity, just as customers might have more than one attribute of interest. The attributes are represented as a series of name/value pairs: for example, Customer ID: 555.1212; Transaction: 5. The characters to the left of the colon is the attribute's name, and the number to the right is the attribute's value. Multiple attribute/value pairs might exist for every entity.

II.1.c. Creating and Disposing Entities. Entities are dynamic objects in a process, or network oriented, simulation. New entities are created at runtime and undergo a process. Once the process described by the network block diagram is complete, the entity is considered to have left the system. The process of an entity leaving the system should be explicit so that statistics of interest, such as the time the entity spent in the system, can be collected.

The space taken up by the entity should be reclaimed by the system when the entity departs. The need to reclaim space is not academic, since the representation of an entity takes up computer system resources, including main memory. A simulation with thousands of undisposed entities might run out of computer resources and cause a program error.

II.1.d. Queues and Resources. In many systems entities compete for scarce resources. Not every customer entering a bank can be serviced by a teller immediately, and not every part in a production line can have an operation performed on it by a machine tool without waiting. As a result some entities are forced to enter a queue until the resource becomes available. As can be expected a corresponding situation occurs in the computer model of the actual system.

A resource is defined as an item that is both limited and in demand by entities as they proceed through their defined process. The number of entities demanding a resource may outstrip the resources available; a queue will form as the entities wait for the resource if this occurs. The resource is also called a server, in an analogy to the bank teller example. There may be zero, one, or more servers for entities to seize at any given time, depending on the model employed by the analyst.
Each resource must have an associated queue. If an entity demands a resource and there are none available, it waits until the resource is freed (barring special cases, such as balking, jockeying, or reneging).

The queues have *queuing disciplines* which specify the order in which entities waiting for a resource are satisfied. Some of the more popular disciplines are FIFO (First In, First Out), LIFO (Last In, First Out), random, and priority. In a random queue entities are served in no particular order. In a priority queue the entities are ordered by some attribute that denotes the order in which they will gain control of the resource—perhaps by transaction type in the teller example, or in a factory simulation by the urgency a part is needed.

An entity might *balk*, or refuse to join a queue when it arrives at the service point because it considers the line to be too long. It might also *renge*—exit from the queue after is has joined—or *jockey*—shift among queues in search of a shorter line.

II.1.e. Decision Making. It is useful to allow an entity traveling through a directed graph of network blocks to make decisions about what block it will go to next. For example, a piece part traveling through a simulated production line might have an attribute that marks it as defective, signaling that some sort of repair work is required before it can continue. A decision block could detect the attribute value and route the part to a repair station, where the part could be fixed and returned to the production line. Entities without the attribute would continue through the normal production line.

II.1.f. Data Analysis. A simulation model without data collection capabilities is like a computer without output display: it may do useful work, but there is no way for anyone to know about it if it did. Models are built to analyze the relevant aspects of the real world systems, and data about the model’s performance is needed before any conclusions about the system can be drawn.

Analysts use two types of statistics to measure performance: observation based statistics and statistics based on time-dependent variables. Observation based statistics are the most familiar. They are determined by recording a number of data
values, then determining the mean, mode, standard deviation, and other statistics by the usual methods. Statistics based on time-persistent variables depend on two numbers: the value of a variable, and the time at which the variable state changes. Pritsker and Pegden (1979) discuss the distinction and provide equations for calculating descriptive statistics.

*Ents* uses the standard $S^2$ statistic is used as a variance estimator for all statistics based on observation. This is a severely biased estimator for data that is correlated. Unfortunately, many common simulation statistics are highly correlated. The average waiting time in a queue and the average time spent in a node are two examples (Kelton and Law, 1991). A long waiting time for one entity is likely to indicate that the next entity will also wait for a long time; this fact manifests itself as a positive covariance in the data and destroys the assumption of independence necessary for a unbiased $S^2$ statistic. The actual variance is likely to be much larger than that indicated by $S^2$. While corrected variance estimators are available, they generally have poor performance due to the difficulty in estimating the correlation factors for data from unspecified distributions (Kelton and Law, 1991).

*Ents* follows a widespread industry convention in reporting the misleading $S^2$ statistic rather than a corrected value. In this case consistency with past practice seemed more important than strict theoretical rigor, particularly when even the corrected values are often poor estimators of the actual variance. One way to skirt the problem of correlated output data is to replicate several runs, and then use the observed (and statistically independent) output values to calculate another $S^2$ statistic. This statistic can then be used to place confidence bounds on the output parameter.

Some of the more popular statistics used to determine model performance are the utilization of resources, the average time an entity spends in a queue, and the average time an entity spends in the model. The first is based on a time-persistent variable, the last two upon observation based statistics. It is also often useful to collect statistics on some specified entity attribute that might represent, for example, the number of times a part was machined by a particular piece of equipment.
II.2. Some network topology considerations.

II.2.a. Defining process classes. Deciding just how to group discrete events into process classes greatly influences the character of a simulation tool and the grammar of valid network diagrams.

Just as not all possible arrangements of words can form a syntactically correct sentence, not all possible arrangements of process classes can form a valid network diagram. A simulation element that represents a queue is meaningless if no entity is able to reach it, or if no destination is specified once an entity finishes service. In such a situation it is desirable to alert the user to the error and disallow execution of the model. The model cannot have been correctly specified, and no purpose is served by attempting to run it.

It is desirable to define the process classes or network nodes so that they are position-independent—any node may follow any other and connect to any other without regard to the type of nodes involved. This is consistent with the project goal of making an easy-to-use simulation tool that does not require extensive a priori knowledge of a complex syntax.

A network formed by a set of network block definitions that are completely position-independent is valid if each block has a place from which it receives entities and has a place to which send entities are sent once they have been processed. So long as this condition is met, the network is syntactically valid. Note that this criterion allows disjoint directed graphs in one model—two or more networks may exist that have no connection to each other. Also, only obvious syntax errors can be caught by the program, not semantic (or logical) errors. If a queue node has its inputs and outputs correctly specified but decision making nodes route all entities away from the queue node in question the program will not warn the user of an error condition.

An alternative approach is to define process classes that are position-dependent, i.e., nodes that require another node of a specific type to be present elsewhere in the directed graph's flow. A complete check for syntactical correctness in such a network is more difficult than in a position independent
network, since we have to "walk the graph" in some sense to determine the types of nodes that occur downstream from a position dependent process class. One technique is to find the network's transitive closure, an operation that determines the set of nodes that can be reached from any particular node in a directed graph. With the information added by this operation a specific node's grammatical correctness can be checked by confirming that an entity departing from the node actually does eventually reach a matching node of the required type.

It has been proven that the transitive closure of a directed graph can be found in time no worse than proportional to $V^3$ for a dense graph or $V(V+E)$ for a sparse graph, where $V$ is the number of nodes and $E$ is the number of edges connecting the nodes (Sedgewick, 1988). A dense graph is common in simulation network models. The transitive closure need be determined only once during each check of the directed graph; the data structure that results can be used for all subsequent examinations for syntactical correctness of a particular node.

The question of position-independence most notably arose in the handling of queues and resources. One way to define the process classes for resources and queues is to separate them into distinct queue blocks, resource allocation blocks, and resource deallocation blocks. Entities arrive at a queue block, then proceed to a seize block (which must immediately follow the queue block) when a resource is available. The entity may undergo a series of operations or seize other resources before arriving at a release block, where the original resource is freed and made available to service other entities. This offers a great deal of flexibility, but somewhat more difficulty in determining when the user has specified an obviously incorrect model, and shifts a larger share of the cognitive load onto the user. The user must know a seize block must follow the queue block and several other rules of syntax, knowledge that may be difficult to convey in the system image. The simulation tool ought to place minimal demands on the user and warn him if the models he creates are likely to be incorrect. Both goals are more difficult to achieve when the syntax for connecting nodes is not position-independent.

An alternative to having separate queue, seize, and release network blocks is to combine all three into one block. An entity would arrive, wait if needed, seize a resource, and undergo a delay representing its service time. When its service time has been completed, the entity releases the resource and continues on in the
network. Since all the operations are encapsulated in one block it is impossible to create an invalid arrangement; the resource is always released at the end of the service activity. It greatly relaxes the restrictions on network grammar, since it is no longer necessary to immediately follow a queue node with a seize node, and the release block cannot be misplaced. If other network blocks are position-independent, the directed graph can be checked for syntactical correctness by simply confirming that all the blocks are completely connected, or equivalently, that no block has an inlet or outlet that is not connected.

The algorithm that determines syntactical correctness of this type of network is proportional to the number of nodes in the network (worst case), compared to $V^3$ for the approach that uses separate blocks.

The latter arrangement, with queue, seize and release functionality combined in one node, was the one used in *Ents*, the program developed for this project. Each approach has its advantages—a composite queue, seize, and release block is easy to implement and is simpler to understand, while separating them into distinct blocks allows entities to seize multiple resources and to undergo multiple processes while in control of a resource. Defining separate queue and resource process classes would let the modeler simulate a wider range of problems, but it was decided that the benefits of relaxing the requirements for a network's grammar and reducing the programming load required to implement the simulation tool outweighed the added flexibility.

II.2.b. Topology changes on running simulations. A Visual Interactive Simulation tool has to face some problems not encountered by batch simulation programs. A simulation might run for a while, be stopped by the user, have parameters or even the basic network topology changed, and then continue running. Stopping the simulation returns the user to the model building phase—in theory, a completely new model with a radically different topology can be built when the simulation stops. In contrast, batch simulation tools keep the same model for the duration of the run.

Allowing the user to change the network topology presents some practical and philosophical problems. Most discrete event simulations use a so-called *event*
list to keep track of pending system events. The type of event, entity, time, and the block it occurs at are kept in a priority queue, then removed and dispatched as the system clock advances to the time of the scheduled event. If an event is scheduled to occur at a particular node and that node is deleted by the user, the model enters an undefined and possibly unstable state. Entities may disappear from the system in an unpredictable way and the validity of the model becomes highly questionable.

For example, removing a queue node while the model is running might also effectively remove any entities waiting in line for service. Entities that had been scheduled to arrive at the node would be lost when the event is dispatched by the simulation executive, if program integrity can be maintained at all. Changing the layout of the network blocks by modifying the connecting arcs leads to a similar situation. Extensive changes by a user might bring into question the relevance of the modified model vis a vis the original layout.

It is conceivable that a change to the topology of the network could be gracefully managed by requiring the user to review the status of the event list and of entities affected by the changes. Implementing such a capability could become quite involved, to say the least.

Another option is to disallow any changes to the network topology once the simulation has begun running. Changes may be made only when the event list is empty and are not allowed at any other time.

Merely changing the parameters of the network nodes creates fewer problems. An activity duration might be changed from 10 minutes to 5 minutes without causing an obviously inconsistent event list. Entities currently in the activity would exit normally and continue to the next node after their allotted 10 minutes, while new arrivals to the node would have an activity duration of 5 minutes. Allowing entities to complete their originally scheduled service time may or may not be what the user intends, but it is at least a reasonable viewpoint to enforce.

The program developed for this project allows no changes in network topology once the simulation has begun running, but does allow network node parameter changes during simulation execution.
II.3. Support functions.

A simulation engine needs several utilities that perform fairly mundane but necessary operations. They are usually complex or pitfall-prone enough to require non-trivial analysis and coding, but are not central to the program logic.

II.3.a. Random number generation. Activity length, the time between entity creations, some attribute value assignments, and many other aspects of the simulation engine rely on algorithms that produce random sequences of numbers, or rather sequences that appear to be random when analyzed by standard statistical tests.

The program developed for this project, *Ents*, uses three pseudo-random number streams to simulate the random aspects of the systems it models. All are based on the linear congruential method, $Z_i = (aZ_{i-1} + c) \mod M$, a technique that has been extensively examined in the literature and is widely used. See Law and Kelton (1991) for a discussion of several random number generators.

The parameters for the random number generators used in the program are shown in Table 1.

<table>
<thead>
<tr>
<th>Stream</th>
<th>m</th>
<th>a</th>
<th>c</th>
<th>Initial Seed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2147483399</td>
<td>40692</td>
<td>0</td>
<td>42</td>
</tr>
<tr>
<td>2</td>
<td>2147483563</td>
<td>40014</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>2147482811</td>
<td>41546</td>
<td>0</td>
<td>69</td>
</tr>
</tbody>
</table>

These generators are recommended by L’Ecuyer (1986) on the basis of their spectral properties from dimensions 2 through 6 after an exhaustive examination of the 50 largest primes $m$ smaller than $2^{31} - 1$ and $a$ within certain specified conditions. Interestingly, L’Ecuyer found that the popular IMSL parameters $m = 2147483648$ and $a = 16807$ were outperformed by the above generators in this respect.
The random number streams are shared among all objects that request random numbers. A queue node that requests a service time might take random number \( N \) from a stream; an activity node in the same simulation model could take random number \( N + 1 \) from the same stream. This prevents one type of unwanted cross-correlation that would occur if all the nodes used private streams with the same starting point. It would be undesirable, for instance, to have each node generate a similar if not identical random number at each step of the simulation. An alternative to this approach is to specify a unique random number seed at each node, perhaps for each random process in the node (e.g., servers in a queue). In fact, this can be a very useful feature during the model output analysis phase when certain variance reduction techniques can exploit the induced covariance of the output parameters. However, it was felt that the inclusion of such a feature would be an unwarranted assumption regarding the level of sophistication in the user base and would unduly complicate the interface for beginners.

II.3.b. Random variate generation. The random number generation techniques discussed above simply return a pseudo-random number uniformly distributed between 0 and 1. A simulation engine will also need random variates from specific distributions. Law and Kelton (1991) discuss the theory and practice of random variate generation for specific distributions.

Ents allows selection of several of the more popular random variable distributions from the pop-up lists and forms that assign random numbers, including exponential, normal, uniform, and fixed distributions. Providing many random variable distributions was not an overriding concern for this effort. However, adding new distributions is not overly involved if the source code is available.

The fixed distribution always returns the single parameter specified. The term "random variable" is something of a misnomer in this case—the distribution is deterministic.

The uniform distribution returns a number between the upper and lower bounds with equal probability. The program uses the inverse transform method to convert a uniform \((0,1)\) deviate into a number from the uniform\((\text{lower, upper})\) distribution. If the user reverses or mislabels the upper and lower bounds and the
parameter labeled as the "upper" bound is actually smaller than the other parameter, the generator will reverse the parameters before generating the random variable.

The exponential distribution is a popular single parameter distribution. The program uses the inverse transform method to map a uniform(0,1) deviate into the new distribution.

The normal distribution is another popular random variable distribution that *Ents* can utilize. The polar method is used to create new normal deviates from the random number stream.
III. System Description

Creating and running models with *Ents* is fairly straightforward after some basic concepts have been explained.

III.1. The opening window.

When the application starts up there are two windows present: the simulation window and a palette that contains symbols for various network blocks. Figure 1 shows a typical arrangement. The main menu (not shown) is typically placed at its default spot, the upper left hand corner of the screen.

![Figure 1. The opening screen.](image)

To add a node to the model, click on the appropriate symbol with the mouse and drag it over to the simulation window, then release the button. The node will be added to the window. If you attempt to release the node in an inappropriate area—the background or the window of another application, say—the node will appear to travel back to the palette and no action will be taken.
III.2. Network symbols.

III.2.a. *Ents* defines eight network blocks from which simulation networks can be built. The model is created by placing blocks into a window, connecting them by drawing lines with a mouse, and then setting parameters for individual nodes. A network block that represents an activity is shown in Figure 2.

![Activity Node](image)

Figure 2. A network block

Each block type has its own icon; in this case, a clock face that represents the passage of time. The block’s name is below the node. It can be edited by clicking on the text with the mouse and typing in new information. The node can be dragged to a new location on the background by clicking on it with the mouse and then, with the mouse button still down, moving to the desired location.

The parameters of the network node, such as the activity duration and the random number stream, can be viewed and set by double clicking on the icon. A panel to display and set the various network parameters will come up, and the user can select and specify values in the normal NeXT Step ways.

This node has two *connectors*, the small black boxes on the right and left of the icon. Entities arrive through the connector on the left hand side, and are sent to the next node through the connector on the right hand side. No source or destination nodes are shown for the node in figure 2.

To specify the next node an entity will go to, click down with the mouse somewhere in the right hand connector box and then drag, with the mouse button still down, to another connector. A line is drawn from the connector box to the current mouse location to let the user know something is happening.
When the mouse is over another connector box that is capable of accepting the entity, the receiving connector box will highlight. If the user releases the mouse button while over the accepting connector the line will attach to it; if not, the line will disappear. The line signifies that an entity path exists between the two nodes.

A connector is either an inlet connector or an outlet connector. An outlet connector specifies what node to send an entity to next. It may have one and only one line connecting it to another node, since the entity can be sent to only one node. Once a line has been drawn from an outlet connector no further associations are allowed. The user must first delete the existing link and then create a new one.

An inlet connector can have many lines connecting it to other nodes, since many blocks might need to send entities to the same node.

Outlet connectors must be linked to inlet connectors, and vice versa. This is just common sense—it serves no purpose to have two outlets connected to each other. The program will disallow any attempts to make such a connection.

The concepts above are applicable to all the process classes defined in Ent$ts$. Specific information on what the nodes do is described below.

III.2.a.i.Create nodes. The create nodes manufacture new entities at a user-selected interval. The symbol for the create node is shown in Figure 3.

![Create Node](image)

Create Node

Figure 3. The create node.

The create node only sends entities on to other nodes and does not receive them. It has one outlet connector but no inlet connector.
The inspector panel for the create node is shown in Figure 4. As with the other nodes, the inspector panel is called up by double clicking on the icon of the block in question in the simulation window.

Figure 4. Create node inspector panel.

The inspector uses the usual NeXT Step conventions, including the close box in the upper right hand corner, buttons, fields, and pop-up lists. Fields that cannot be edited are denoted by light grey text. These only display information and cannot be changed.

By default the node creates an entity every time unit, though the parameter can be changed by editing the field at the top of the window. To select a random number distribution other than Fixed, click on the button in the Time Between Creations box and then drag to one of the other distributions. Exponential, normal,
fixed, and uniform distributions may be selected from the list. The names for the
distribution's parameters will appear in the form to the right, where the user can
enter new values. The random number stream can also be selected in a similar
manner from another pop-up list.

Negative values for time between creations are meaningless, and are therefore disallowed. If the user attempts to enter a negative number he will be
alerted by a beep and the cursor will not advance to the next field. Restrictions on
other random number parameters are also enforced so that the user cannot enter
negative values for the normal distribution's variance or other obviously incorrect
data. Time duration random variables are not allowed to have negative outcomes.
For example, suppose the user specifies time between creations to be a normal
distribution with mean 1.0 and variance 1.0. A certain percentage of the results will
be less than zero, clearly a meaningless result in this context. The random variable
will be rejected and another generated automatically; effectively, the tail of the
distribution will be truncated and redistributed over the remainder of the probability
density function.

If the reporting check box is on, the node's summary statistics will be
printed to a window if the user decides to print a report. If node statistics are of no
interest, switch the check box off by clicking in it.

The create node can stop creating entities after either a specific time in the
simulation or after a certain number of entities have been created. To enable either
option, click on the appropriate check box and then edit the field in the form below
it. By default there are no limits on entity creation.

Any comments or notes about the node can be entered in the scrolling text
field at the bottom of the panel.

Once the create node parameters are set to your satisfaction, click on the Set
button in the lower right corner of the panel to save the settings. The Revert button
restores the node to the state it was in the last time the Set button was pressed. *If the
inspector panel is closed without pressing the Set button the parameters will not be
changed.*
A count of the entities created and the total entities created are displayed in the form. The *Entities Created* field refers to the number of entities created since the statistics were last cleared, while *Total Entities Created* is the running count of entities created since the simulation began. It is the *Total Entities Created* field that the entity creation limit refers to.

III.2.a.ii. The destroy node. The destroy node is the counterpart to the create node—where the create node makes instances of new entities, the destroy node disposes of them and frees the computer system resources taken up by the entity representation. A destroy node does not forward entities to any other node, and therefore has only an inlet connector. The icon for the destroy node, the late, lamented black hole from release 1.0 of the operating system, is shown in Figure 5.

![Destroy Node](image)

Figure 5. The destroy node.
The inspector panel for the destroy node is shown in Figure 6.

![Destroy Node Panel](image)

**Figure 6.** Destroy node inspector panel.

The destroy node inspector panel is quite similar to the other inspector panels and operates in an identical way. The *Disposed* and *Total Disposed* fields refer to the number of entities received since the last time the statistics were cleared and since the simulation began, respectively. Some statistics on the entities disposed by the system are shown in the *Time In System* form in the middle of the panel. The mean, variance, minimum, and maximum amount of time the entities disposed by this node have spent in the system are shown in fields that cannot be edited; therefore the field name and contents are greyed out.
The simulation can be stopped after a specific number of entities have passed through the destroy node by selecting the *Limit Disposals* check box. The *Disposal Limit* field will become active and a new value can be entered. When the disposal limit has been equalled by the total number of entities disposed the simulation will stop execution.

III.2.a.iii. The assign node. Entities are objects which travel through the network and have attribute/value pairs that describe the entity. Instances of entities are manufactured at create nodes and disposed of (usually) at destroy nodes. The entities are given user-specified attribute/value pairs at assign nodes, the icon for which is shown in Figure 7.

![Assign Node](image)

**Figure 7.** The assign node.

Entities have a few predefined attributes assigned by the system and an unlimited number of user-defined attributes. The system-defined attributes cannot be redefined or assigned new value, but user-defined attributes can be added or changed at will. The restriction on system-defined attributes ensures that they always return the values expected of them.

The attribute names are text strings that can be any length and can include spaces or other characters; capitalization is not important, so “This Attribute” is equivalent to “THIS ATTRIBUTE” or “tHiS attRIBute.” Though case is not important, the program preserves the capitalization originally used to enhance readability. Any text string can be used as an attribute name, but it is unwise to use text that might be interpreted as a number, such as “12” or “1.4”, for reasons that will be explained later.

Three attribute names are reserved and will return only predefined values. They cannot be redefined to be associated with other values. They are: “Time In
System”, “TNOW”, and “Creation Time”. The first always is always associated with the time the entity has spent in the system since creation; “TNOW” is associated with the current simulation clock time; and “Creation Time” is paired with the time at which the entity was created.

The values associated with an attribute name are in all cases double-precision floating point variables. For example, an entity might have attribute/value pairs as follows:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time In System</td>
<td>10.4</td>
</tr>
<tr>
<td>TNOW</td>
<td>145.7</td>
</tr>
<tr>
<td>Creation Time</td>
<td>135.3</td>
</tr>
<tr>
<td>Repair Trips</td>
<td>2.0</td>
</tr>
<tr>
<td>Regular Trips</td>
<td>5.0</td>
</tr>
</tbody>
</table>

The first three attributes, “Time in System”, “TNOW”, and “Creation Time”, are system-defined. “Repair Trips” and “Regular Trips” are user-defined attribute/value pairs. The modeler is responsible for assigning and updating values for these attributes. “Creation Time” is set once for each entity and does not change during the entity’s lifetime. The “Time In System” and “TNOW” values are updated automatically by the program as time passes; no user intervention is needed to obtain correct values for these attributes.

It should be emphasized that the attribute/value pairs are unique to each entity. Another entity might have different attributes and values, and changing the value of an attribute in one entity has no effect on the attribute in another entity.
The inspector panel for the assign node is shown in Figure 8.

![Figure 8. Assign node inspector panel.](image)

The name of the attribute is entered in the field at the top of the panel. The user selects which of several values to assign to the attribute based on the state of the so-called "radio buttons" at the top left of the panel.

To assign a random variable to an attribute, click on the random variable radio button. All the other radio buttons will switch off and the selected button will switch on, just as a push-button car radio works when selecting a new station. Enter the parameters for the random variable in the field, and then make the selection permanent by pushing the Set button.
The current system time—the value of the system clock when the entity arrives at the node—can be assigned by clicking on the current time radio button. The attribute will retain the same value until it is reassigned, unlike the TNOW system assigned variable, which always reflects the current simulation time.

The attribute can also be assigned the length of a queue or the number of queue servers available. When either of these radio buttons is pressed, the Queue Name field becomes active. Enter the name of the queue you want to monitor. (The queue name is the text that appears underneath the node icon.)

The queue must already exist, and must be uniquely named. The inspector panel will refuse to accept a queue name that does not exist or which is used by two queues. The assign node is also checked at runtime for ambiguous name references; if another queue with the same name has been added after assignment node was specified, an alert panel will warn the user and prevent the program from running. Queues that are not mentioned in assignment nodes are allowed to have identical names.

The Queue Length and Servers Available assignment options are useful primarily for decision making. Often the entity will be routed to a different network branch when all the servers are busy or when the line exceeds a certain length.  

When an entity arrives, the node checks to see if the attribute already exists. If it does, the attribute’s value is replaced by the value shown in the inspector window. If the entity does not have the attribute it is created automatically and added to the list of attribute/value pairs. There are no limits on the number of attributes an entity may have, aside from computer system restrictions.

III.2.a.iv. The activity node. The activity node delays an entity by a specified amount of time. Any number of entities may be undergoing an activity at one time. An example of an activity might be the delay associated with moving a part from one station to another via conveyor belt. There is essentially no limit on the number of entities that can be engaged in the activity, and the process takes some amount of time to complete.

Routing to a different node when the line becomes too long can also be achieved with the balking feature of queue nodes.
The icon for the activity node is a clock face, shown in Figure 9. An activity node has one inlet connector on the left hand side of the node icon and one outlet connector on the right hand side.

![Activity Node](image)

Figure 9. The activity node.

The inspector panel for activity nodes is shown in Figure 10.

![Activity Node Inspector Panel](image)

Figure 10. Activity node inspector panel.
The length of the activity can be set by selecting a random number distribution from the pop-up list and then entering the distribution’s parameters. Either constant numbers or entity attributes can be entered as random number distribution parameters. If the latter option is used, the entity will be examined and the value of the attribute looked up and used as the parameter. If an entity arrives without the attribute shown or a value that is obviously invalid (a negative variance for the Normal distribution or a fixed length activity time of less than zero, for example), the simulation will stop with an error message.

Constant numeric values and entity names are distinguished by first attempting to resolve the data entered into the field as a number. If a negative number is entered the field will disallow entry; if a positive number is entered the field will accept the data. If the data cannot be interpreted as a number it is assumed to be an attribute that will be present in all entities that arrive at the node. It is for this reason that giving an attribute a name that might be construed as a number is a bad idea; an attribute named “13” would always be resolved as the value 13, and the value associated with the name would never be referenced.

The random number stream used by the random variable generator is selected through the other pop-up list in the box.

Reporting for the node can be turned on or off via the labeled check box.

Assorted statistics and status information on the node are displayed in greyed-out text, since they reflect a system state that cannot be changed by the user. The number of entities currently engaged in the activity is shown near the top of the panel, and statistics on the time the entities have spent in the activity and the level of the activity, or average number of entities occupied in the activity over time, are shown in two forms near the bottom of the panel. The latter is a statistic based on a time-persistent variable.

III.2.a.v. Branch nodes. Branch nodes route the entity based on the values of its attributes. For example, all entities that have a time of creation less than 500.0 might be sent to one branch of the network, while those created after that time would be sent to a branch that did special processing for late arrivals. The icon for the branch node is shown in Figure 11.
The branch node has one inlet connector and two outlet connectors. The upper outlet connector routes the entity to one branch if a boolean test is true, and the lower connector routes the entity to another branch if the test is false.

The inspector panel for branch nodes is shown in Figure 12.

![Branch Node Inspector Panel]

**Figure 12.** Branch node inspector panel.
The branch node performs a single boolean test on the value of an attribute given by the user. The name of the attribute is entered into the field at the upper left hand side of the panel, while the value it is being compared against is entered in the field to its right. The comparison operator, one of =, <=, >=, >, <, or <>, is selected from the pop-up scrolling list between the two fields.

The value on the right hand side may be either a constant value or another attribute contained by the entity. For example, the user might specify a left-hand attribute of TNOW, a comparison operator of <, and a right hand value of 550. If the current clock time is less than 550 the entity will be routed to the true branch. Alternatively, the right hand field could contain another entity attribute. The value of the right-hand attribute will be compared to the value of the left-hand attribute, and the entity routed accordingly. Suppose that an entity has been assigned attributes of “Max line length” and “Current line length,” and that the comparison operator is “>=”. Entering the names of the attributes in the left-hand and right-hand fields will ensure that the entity is routed to the true node when the value contained in “Max line length” is greater than or equal to the value contained in “Current line length.”

The inspector panel differentiates between values and attribute names in the same manner that the activity node does: by first attempting to resolve the entry as a number, and then by assuming it is an attribute. Therefore attribute names that might be interpreted as numbers should not be used.

An ambiguity exists if an entity arrives at the branch node without one or both of the attributes referenced in the boolean statement: how can we determine the truth of a statement if we have no data upon which to base a comparison? As a safety measure the user can state what should occur if this situation occurs. The simulation can stop, the default state; the entity can take the true branch, or the entity can take the false branch. Any of these options can be selected by clicking on the appropriate radio button.

Statistics are kept on the number of entities arriving at the node, taking the true branch, the false branch, and default branches (an entity arrived without an attribute).
III.2.a.vi. The arithmetic node. The arithmetic nodes perform arithmetic operations on attribute values. An attribute might have one added to its pre-existing value whenever it comes to the node, for example, or it might be multiplied by another entity attribute. The icon for the arithmetic node is shown in Figure 13.

![Arithmetic Node](image)

Figure 13. The arithmetic node.

The arithmetic node has one inlet connector and one outlet connector. The inspector panel for the arithmetic node is shown in Figure 14.

![Arithmetic Node Inspector Panel](image)

Figure 14. Arithmetic node inspector panel.

As with the assignment node, the name of the attribute is put into the field at the upper left of the inspector panel. The operation to be performed on the attribute,
+, -, /, or *, is selected from the scrolling pop-up list button. The attribute that the result will be placed in is entered into the left hand field, and a number or attribute is entered into the right hand field. As before, the value on the right hand side is first resolved as a number; if that operation fails the string is resolved as an attribute name. This allows either constant values or attribute values to be used on the right hand side of the expression.

If the left hand attribute did not exist prior to the entity’s arrival at the node the attribute is created with a value of zero, and then the arithmetic operation is performed; if the right hand attribute name does not exist in the entity its value is assumed to be zero.

Suppose an arithmetic node is specified such that “Loopcount” is in the left hand field, the operator is “+”, and the number “1” in the right and field. If an entity arrives without the attribute “Loopcount,” it will be created and added to the entity’s attribute list with an initial value of zero. Then the operation will be performed; in this case, one will be added to zero, and the attribute will have a value of one. If the entity returns to the node the value of “Loopcount” will be incremented by one each time from its existing value: 2, 3, 4,...

III.2.a.vii. The statistics node. The statistics node collects summary data on a given entity attribute. The icon for the statistics node is shown if Figure 15.

![Statistics Node](image)

Figure 15. The statistics node.

The statistics node has one inlet connector and one outlet connector. The inspector panel for the statistics node is shown in Figure 16.
The attribute name that observation statistics are to be collected on is specified in the field at the top of the inspector panel. All entities that have an attribute by that name and that pass through this node contribute their data to the summary statistics. If an entity arrives at the node that does not contain the attribute, it is passed on without further action.

The mean, minimum, maximum, count and variance are calculated and shown from those entities that do contain an attribute with the name shown in the form. The statistics can, as usual, be reset to the zero state by pressing the Clear Statistics button.
Occasionally the user might want to confirm that no entities arrived at the node without an attribute by the name shown in the inspector panel. This can be done by comparing the number of arrivals to the node with the observation count in the statistics form. If such an entity passed through the node the number of arrivals will exceed the number of observations in the summary statistics.

III.2.a.viii. The queue node. This process class is one of the most complex and useful node types. Many business simulations become, in effect, a network of interacting queues, activities, and decision actions that are too complex to be solved by classical queuing theory techniques. The icon for the queue node is shown in Figure 17.

![Figure 17. The queue node.](image)

The queue node has one inlet connector and two outlet connectors. The connector at the right of the node is the normal outlet connector, where entities are sent when they complete service. The connector at the top of the icon is the balk connector.

Balking is a user-selected option. If an entity arrives and finds that the line is too long it immediately balks and goes to the node pointed to by the balk connector. If balking is not enabled the connector is not required to point to another node. This is the only exception to the rule that all connectors must point to another connector.
The inspector panel for the queue node is shown in Figure 18.

The entity will arrive at the node and check for an available server. If no server is available the entity goes into a line and waits until one becomes free, then enters service. The time the entity spends in service is a random variable; afterwards the entity is sent on to the next node.
Balking is enabled by clicking in the marked check box. The maximum line length field becomes active, and the user may enter the largest size the line can grow to. For example, a drive-up teller window might have physical or logical constraints on the number of cars that can wait for service at one time: limited space for cars in the line or impatient customers, perhaps. If balking is enabled the balk output connector must point to another node. An error is reported by the program when it attempts to run if no balking connection exists and balking is enabled.

The length of service random variable is selected in the usual way from the pop-up lists and field entries in the box at the top of the inspector. The queue discipline can be selected from the pop-up list to the left; the options are FIFO, LIFO, and Priority. If the priority option is selected the priority attribute field is activated in the form below it and the user is allowed to enter the attribute name.

If the queue is in priority order the line is arranged in descending, FIFO order: the entities that have the highest value associated with the priority attribute are placed first in line, and if any ties occur they are broken by putting the later arrival further back in the line. If an entity arrives at a priority queue node without the attribute the line is ordered by, the entity is placed at the back of the line in FIFO order. New entities with the attribute will always be placed in front of older entities without the attribute.

The number of servers is set through the Servers field, which can be changed to any non-negative integer. If servers are added while a line for service exists, the waiting entities are taken in order from the line and placed into service immediately. If the number of servers is reduced below the number already busy the servers will be removed when the entities they are occupied with finish service.

The number of servers that are busy and the length of the current line are shown in greyed-out text to signify that the user cannot directly change these parameters.

Several summary statistics on the performance parameters of the queue can be shown in the two forms towards the bottom of the inspector panel. Observation
based statistics are displayed in the left hand form and statistics based on time persistent variables in the form to the right. Each form can display statistics for a number of variables by selecting the pop-up list above the form. Observation based statistics that are collected automatically include Time in Node, Time in Line, Time Being Serviced, and Time Between balks. Time in Node refers to the cumulative waiting time and service time, while Time in Line and Time Being Serviced are a more detailed breakdown of the same performance data. The remaining statistics are self explanatory.

The statistics based on time-persistent variables are displayed in the form to the right. These include Server Utilization, Entities in Line, and Entities in the Node.

Server Utilization is the average number of servers busy, over time. This is particularly useful information for use in determining how many servers are required to efficiently serve customers. High utilization is the usual design objective when optimizing a model, but this also has a side effect of increasing line length and customer waiting time.

The average number of entities in line and the average number in the node are two other summary statistics that are useful in optimizing models. The Entities in Line statistic refers to the time-averaged number of entities waiting for service, generally a parameter that should be minimized. The Entities in Node statistic is the time-averaged number of entities both waiting and undergoing service.

III.3. Running simulations.

Building and running a simulation is simple. A typical window arrangement is shown in Figure 1. Drag the node objects over from the palette, drop them into the simulation window, and then hook up the objects by dragging between connectors with the mouse. The simulation window can be made larger to view models that take up more space, and the background upon which the network nodes rest can be scrolled with the sliders on the edge of the window. The background will scroll automatically when the user drags an existing node beyond the right hand or upper edges of the visible region.
All the connectors, with the possible exception of one or more queue node balk connectors, should point to another node. Enter the time to stop the simulation by typing in the text data entry field in the upper center of the simulation window, and then press the Run button. The simulation will execute until it reaches the end time or until one of the nodes in the model meets the simulation termination criteria.

The program checks for obviously incorrect data before running the simulation. If a node has incorrect or incompletely specified data an alert panel will pop up with the name of the node and a description of the problem.

The simulation window is in a so-called modal loop when the model is running and ignores input to menus, panels, or windows other than the running simulation window. The active simulation window still accepts input, however; the simulation can be stopped by pressing the Stop button, a node inspector panel can be opened, and the speed of the animation can be changed by moving the slider (animation and other features are discussed below). Allowing a model to run in background while the user continues to work in other windows would require multiple threaded execution of the application. The NeXT supplies the tools to do this, but this feature has not been implemented in the current version of the program.
The application’s main menu is shown in Figure 19. For the most part the menu contains standard NeXT Step items that will be familiar to any NeXT user.

The menu items with arrow heads to their right signify submenus, and those ending with a “...” signify that a dialog box or some other panel will come up when the item is selected. Submenus can be “torn off” from the main menu and made into stand alone menus; all the menus can be dragged to more convenient locations on the screen. Most NeXTs are set up to place the main menu in the upper left hand corner of the screen when the application starts.

The menu that is most concerned with simulation proper is the Simulation submenu, a submenu of the main menu. It is shown in Figure 20.
The Run and Step menu items are alternatives to the buttons in the simulation window. The have keyboard equivalents of command-r and command-N, respectively. Run begins simulation execution; the model will run until it is either stopped by the user or it triggers some termination criterion, such as simulation length or the number of entities that have exited the system. The Step menu item advances the model by one event.

A textual summary of the model’s performance can be shown in a window at any time by selecting the Report menu item. All nodes that have the reporting check box enabled in their inspector windows will have their relevant statistics summarized and tabulated. Animation and execution tracing are discussed in the debugging section below.

Many models have a so-called “warm-up period” during which the model reaches steady-state conditions. The statistics gathered while the model is in the transient state can bias the estimate of the true steady-state system behavior, if ascertaining steady-state behavior is in fact the objective of the modeler. The statistics gathered by each node can be cleared by bringing up the inspector panel and pressing the Clear Statistics button, but doing this for each of the nodes in a large model would be very tedious. The Clear Statistics... menu item offers an alternative. All the node statistics can be cleared at once by selecting the Clear Statistics... item from the Simulation menu. An alert panel will come up to confirm the user’s intent and then continue with the operation.
The model can be reset to its original, time zero state by selecting the *Reset...* item from the *Simulation* menu in a similar manner. It is necessary to reset the simulation to make topology changes or to add nodes to the model. Changing the parameters of existing nodes does not require a reset.

If the palette window disappears, a new one can be opened or the old one brought to the front by selecting the *Palette...* item from the main menu.

The *Trace Execution* and *Animation* menu items are discussed below with other information related to debugging.

The above submenus and menu items are the only unique additions to the main menu. The remaining menus and menu items are standard NeXT Step application features which should be familiar to most users. For example, the model can be saved to disk at any time by selecting the *Document* menu item. This will present a standard NeXT Step save menu, which can call up the usual file saving and opening panels. The model will be saved with its current state—including the current simulation clock, event list, and node statistics.

The *Windows* menu item in the main menu shows all the application’s open windows in a submenu. The *Windows* menu is another of the standard NeXT Step menu items that should be familiar to NeXT users.

The *Services* menu provides access to NeXT “Services,” or cooperating programs that provide some utility function. While the services available will differ from system to system depending upon how the system administrator has set up the computer, it is almost certain that the user will be able to look up words in the Digital Webster on-line dictionary and to send text to another user by electronic mail. All the user needs to do is select an area of text with the mouse and then select the corresponding menu item from the *Services* menu. A program called nxypolot, available from NeXT-related electronic bulletin boards at no charge, can take textual numeric data that has been selected with the mouse and create charts and plots.
III.4. Output features.

*Ents* uses the normal NeXT Step conventions for printing. Simply click on the Print... item in the main menu and a dialog box comes up that allows the user to pick a printer or file to which to send output.

III.5. Debugging.

It is the rare analyst indeed who can always correctly specify the simulation model the first time. *Ents* includes debugging tools that let the user trace through program execution and entity flow.

Clicking on the Trace Execution menu item of the Simulation menu brings up a text window that prints a record of every event executed by the simulation engine. The event time, event type, and name of the node at which the event occurred are printed to the window for every event that is executed. In addition, all the attribute names and values associated with the event's entity will be listed.

The Step menu item in the same menu and the Step button in the simulation window single-step through the event list. Both have the same effect—dispatching a single event and then stopping the simulation for examination. Stepping through the model execution is most useful when the simulation is in the execution tracing mode.

One particularly powerful debugging tool is the animation capability. When animation is turned on from the Simulation menu the movement of entities between nodes is shown by a small sphere that travels from one icon to another. A few minutes examination of a running simulation is often all that is needed to determine that a model is routing entities incorrectly. The animation can be sped up or slowed down by adjusting the animation speed slider.
IV. Application Examples

The specifications of the simulation process nodes were discussed in detail in the previous chapter, but no simulation is made of nodes working in isolation. At least a few nodes need to be used in a typical model, and increasing the scale from single nodes to an interconnected network presents hurdles that users often find difficult to overcome. This chapter gives two examples of modeling entire systems with *Ents*: a TV repair line and a Quarry operation, two examples that have been used throughout the literature.

IV.1. TV repair problem.

One of the canonical examples used in simulation texts is that of a TV repair line, originally developed by Schriber (1974). Pritsker and Pegden (1979) restate the critical parameters of the system and model it using SLAM. Pritsker and Pegden's description is repeated here, and then the model will be developed step by step in *Ents*.

A television inspection and repair line has units arrive to its entry point at a rate uniformly distributed between 3.5 and 7.5 minutes. Two inspectors work at the entry point side by side, each examining sets as they arrive. An inspector takes between 6 and 12 minutes, uniformly distributed, to check over a TV. If the set passes scrutiny it is sent on to the packing department and out of the system of interest, but if it fails it is sent to an adjustment station where another worker tunes it. Once that operation has been completed the adjusted set is sent back to the incoming inspection station. On average 85% of the units pass inspection and 15% are in need of an adjustment; the adjustment takes 20-40 minutes, again uniformly distributed. Sets that have been adjusted are just as likely as new arrivals to be found defective. A schematic diagram of the system is shown in Figure 21.
Figure 21. TV Inspection and Repair Line schematic diagram.

The TV sets arrive, are inspected, and either sent out of the system or to the adjustment station. At the adjustment station they are fixed, perhaps, and then sent back to the arrival station to be inspected with the other new TVs.

A model of this system is characterized by an entity creation mechanism to correspond to the entry of new TVs into the inspection line; a queue with two servers acting in parallel related to the inspection station; another queue for the adjustment station, this one manned by a single person; and a way to send some of the entities to the adjustment queue and the rest out of the system model. The creation mechanism, queues, and entity disposal parts of the model are easy to implement since they have exact equivalents in the defined process classes. Finding a way to route the entities is slightly more difficult. Two $Ents$ nodes are needed to do this, one to assign an attribute and one to dispatch entities based on the attribute value.
From the opening screen shown in Figure 1, drag over a create node, two queue nodes, an assign node, a branch node, and a destroy node from the palette. Arrange them in the simulation window as shown in Figure 22. More meaningful captions can be placed on the nodes by clicking on the text with the mouse and entering new names.

![Image of node layout](image.png)

Figure 22. Initial node layout for TV inspection.

Using the mouse, specify the routes an entity can take as it travels through the model by connecting the nodes to each other. At the create node, click on the connector box and drag out a line to the inspection queue. When the pointer makes contact with the inlet connector the small box will highlight, signifying that a connection is possible. Release the mouse and the line will become permanent.
Specify the other routes so that the entity travels from the create node to the inspection queue, the assignment node, and the branch node. Draw a line between the true outlet connector and the destroy node, and from the false branch connector and the adjustment queue. From the adjustment queue the entities return to the inspection station, so draw another line from the queue output connector to the inlet connector of the adjustment queue. There is no need to connect any of the queue balk nodes, since balking will not be allowed in this model. The network should resemble that shown in Figure 23. when finished. The network topology has been completely specified.

Figure 23. The connected TV inspection model.

The next step is to correctly set the parameters for the various nodes. The nodes have reasonable default values, but these need to be changed to match the requirements of our this system.
Double click on the TV creation node. An inspector panel in which the time between creations can be set appears. Change it to a uniform distribution of 3.5 to 7.5 minutes.

The inspection queue node needs two servers, but by default it has only one. That needs to be changed. Double click on the inspection queue icon and edit the inspector panel so that two servers are present and the service time is uniformly distributed between 6 and 12 minutes. Click on the Set button to make the changes permanent. The inspector panel should appear as in Figure 24.

![Figure 24. TV inspection queue panel.](image)

The assignment node will put a random variable into an entity attribute and then the branch node will examine the attribute and route the entity accordingly. Since the requirement is that 85% of TV sets are to exit the system and 15% of them are to be sent to the adjustment station, we can assign a random variable from
the uniform (0,1) distribution to the attribute. The branch node will examine the attribute and route the entity either out of the system or to the adjustment queue.

Double click on the assignment node, which is named "Repair?" in Figure 23, and edit the inspector panel so that an attribute named "Branch RV" is assigned a random variable from the Uniform (0,1) distribution. Click on the Set button to make the changes permanent and then close the inspector panel.

Open the inspector panel for the branch node and enter the attribute name "Branch RV" in the left hand side of the form, then put the value 0.15 in the right hand portion of the form. Select the comparison operator "≥" from the pop-up list and then click on the Set button. When entities arrive at the branch node, the value associated with the attribute "Branch RV" will be compared to 0.15; if it is greater, as it should be 85% of the time, the entity will be routed to the disposal node.

Finally, edit the adjustment queue’s inspector panel so that the service time is uniformly distributed between 20 and 40 minutes.

Enter “480” in the simulation window’s End Time field. New models often have subtle bugs that aren’t readily apparent, so—just to be careful—turn on the animation feature so that the paths the entities take are shown on the screen. Under the Simulation menu, click on the menu item that turns the animation on. Then, click on the Run button in the simulation window and watch the simulation execute.

The first 100 minutes or so of model execution time shows that most of the entities are going where expected. According to the specification 15% of the entities should be traveling to the adjustment node, and that looks about right.

Model verification and validation by “just watching the simulation run” is a dangerous habit to get into. It often gives rise to a false sense of security and discourages detailed scrutiny of the model. A serious simulation effort would include a much more in-depth examination of the model’s correctness and its correspondence to the system being studied; however, many bugs and pitfalls that are not readily apparent in a textual simulation tool can be quickly discovered and eliminated by viewing the flow of entities through the network.
Some summary statistics on the model's performance can be viewed by double clicking on the nodes of interest and viewing the inspector panels. When the simulation finishes the inspection node shows that, on average, 1.8 servers were busy, and that TVs waited for just over two minutes to be inspected, with a maximum of 11 minutes. The branch node routed about 12% of its entities to the adjustment station. The 82 entities that exited the system spent an average of 16 minutes being inspected, adjusted, and moved, according to the summaries in the TV Exit node inspector.

This example is not intended to serve as a complete analysis of the system's performance, but merely to illustrate the operation of the program. The statistical results should be viewed with caution since the simulation has not run long enough nor been repeated often enough to obtain statistically valid results. The "warm-up period," the time during which the system has not yet reached its steady state, is probably also skewing the statistics.

To get a text report on the performance of all the nodes, select the Report item from the Simulation menu. A new window will open and a summary of each node's performance will be printed out. The text can be cut and pasted into documents from other applications in the usual NeXT Step way.

IV.2. Quarry operations problem.

Pritsker and Pegden (1979) present another example, this time based on the operations of a quarry, and create a simulation model using SLAM. The same system is modeled here using Ents.

In a quarry, trucks are loaded from three shovels and deliver ore to a single crusher machine. Each shovel has three trucks assigned to it, one with 50 tons of capacity and two that can carry 20 tons. The trucks are loaded at the shovel, travel to the crusher and dump their load, and then return to the same shovel. A schematic diagram of the system is shown if Figure 25.

---

8The exact values for all the results quoted here will vary with the random number streams used and other factors.
The trucks do not all travel at the same speed or get loaded at the same rate. The larger trucks take longer to fill, take longer to dump, and travel more slowly, but carry more ore. The time it takes a truck to perform each step in the operation is shown in Table 2. (Loading and dumping times are exponentially distributed random variables.)

<table>
<thead>
<tr>
<th>Truck Size</th>
<th>Loading</th>
<th>Travel</th>
<th>Dumping</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 Ton</td>
<td>EXP(5)</td>
<td>2.5</td>
<td>EXP(2)</td>
<td>1.5</td>
</tr>
<tr>
<td>50 Ton</td>
<td>EXP(10)</td>
<td>3</td>
<td>EXP(4)</td>
<td>2</td>
</tr>
</tbody>
</table>

The shovels load the trucks on a first-come, first-served basis, but the crusher takes any 50 ton trucks that are waiting before it will unload any 20 ton trucks.
The process can be modeled rather easily with 4 queues and two activities: one queue for each of the shovels and a queue for the crusher, and activity nodes for traveling to and returning from the crusher. The entities in this model are the trucks which travel back and forth from the shovels to the crusher. However, some complexities arise: the trucks should be present at time zero and never exit the system. They must return to the same shovel from which they came, and depending on their size, take differing amounts of time to perform operations. These features require that several nodes be added to the model that perform initialization functions on the truck entities: creating them, assigning attributes that identify them as 50 ton or 20 ton trucks, and so on. A few branching nodes also need to be added to route the entities back to the shovel they are assigned to.

SLAM can add entities to a model at time zero with attribute/value pairs already defined through the use of the ENTRY statement. EntS has no counterpart to that facility, so several creation and assignment nodes need to be used to achieve the same effect.

Two creation nodes will feed into each of the shovel queues. One of the creation nodes will create two truck entities and then stop production, while the other creation node will produce just one entity. Once created the entities will have an attribute that reflects the truck size assigned (twenty ton or fifty ton), and then be assigned an attribute to reflect its assigned shovel: one, two or three. Each shovel will require two creation nodes, for a total of six creation nodes for the entire model.

The truck entities also need to be assigned attributes that correspond to the times for the various quarry operations, depending on their carrying capacity. An attribute for each of the operations in Table 6 is needed; the attributes will be used as parameters to the random number generators in the activity and queue nodes.

The nodes required to initialize all the truck entities are shown in Figure 26. The create nodes have a time between creations of zero and stop producing entities after the required number of trucks of each type have been produced: two for twenty-ton truck nodes, one for fifty-ton truck nodes. The entities are immediately assigned an attribute to reflect the carrying capacity of the truck created, and then assigned a shovel number. The network feeds into a branch node, where the entities
are separated based on truck size: twenty-ton trucks take the upper branch, and fifty-ton trucks take the lower branch. In each branch the attributes "loading time," "travel time," "dumping time," and "return time" are assigned fixed numbers based on the values shown in Table 6. "Loading time," for example, is assigned a fixed value of 5 in the upper branch; the value of that attribute is later used as the expected value in the shovel queue's exponential random number generator.

Once the entities have been initialized the real work of the simulation can begin. The nodes for this portion of the network are shown in Figure 27. (The two inputs to the first branch node are from the upper and lower branches of the initialization phase.)
The network feeds into a single branch node that routes entities based on the "shovel number" attribute. If the attribute has the value of one, it is sent to shovel queue one; otherwise, it is sent to another branch node. In that node if the attribute has a value of two, it is sent to shovel queue two. All the remaining entities must have a "shovel number" attribute of three, so they are sent to the third shovel.

At the shovels the service time is specified to be exponential with a mean of "loading time," the attribute assigned earlier in the initialization phase of the network. All the entities will have a "loading time" attribute with a value of 5.0 for the twenty-ton trucks or 10.0 for the fifty-ton trucks. The node looks up the value associated with that attribute and uses it to calculate a random service length.
When the entity has completed service it enters an activity node. The length of the activity is a fixed or constant value, "travel time," another attribute that was assigned a value in the initialization phase.

The truck entities wait for service at the Crusher queue node. The queue is ordered by the attribute "truck capacity" so that fifty-ton trucks are served before any twenty-ton trucks, and the service time is set to be exponential with a mean of "dumping time." The entities then travel to another activity node, where they are delayed by a fixed amount "return time," then routed back to the original branch node.

To follow the events as they occur, select the Trace Execution item from the Simulation menu. A window similar to that in Figure 28 will be created. As the program runs, each event dispatched by the executive is displayed in the window. At each node, the system clock time, the type of event, and the attribute/value pairs associated with the entity are shown.

![Trace Output](image)

Figure 28. Tracing program execution.
The model's performance can be examined by double-clicking on a node and examining the summary statistics, or by selecting the *Report* item from the *Simulation* menu. The inspector panel for the return trip activity node after the simulation had run for 480 minutes is shown below in Figure 29.

![Return trip inspector panel](image)

**Figure 29.** Return trip inspector panel.

On average, 0.59 trucks are returning to their shovels at any one time, and the trip takes a mean of 1.6 minutes.
V. Implementation

V.1. Objective-C programming.

The NeXT’s primary programming language is Objective-C, a dialect of C with object-oriented extensions. The language can be used at a very high level of abstraction while retaining the C language’s traditional ability to manipulate bits at the hardware level when necessary. The syntax is (subjectively) less of a leap from C than C++ is, but includes some very powerful capabilities, such as the dynamic loading of objects into a running program and late message binding.

Like most object-oriented languages Objective-C allows inheritance, data encapsulation, and subclassing. The classes’ superclass, instance variables, and the new messages it understands or override are specified in an interface file, while the code that does the work is saved in a implementation file; this cleanly separates what the object does from how it is done.

The syntax of the language adds only a few constructs to that of C. Objects are declared to be a new type, id. They are declared just as one would declare an integer or a float:

```c
id    anObject;     /* declare an object */
int   i, j, k;      /* declare three integers */
```

Messages are passed to objects using square brackets:

```c
[anObject thisMessage];
```

Sends the message thisMessage to the receiver anObject. thisMessage should have been declared as a method in the anObject class interface file and defined in the implementation file.

The NeXT compiler can also handle C++ in addition to Objective-C and C. Mixed programs that combine C++, Objective-C, and C within the same project are possible. However, the AppKit (the user interface toolkit that defines the NeXT look and feel) is oriented towards Objective-C.
Ents was written entirely in Objective-C with approximately six man-months of programming effort, including the initial learning process. None of those on the project had prior experience programming on the NeXT.

V.2. Programming the NeXT.

The NeXT has defined style or "look and feel" that all applications should follow. The location of window controls, the window frame styles, and even the order of the main menu items are specified by NeXT, and all applications should follow the guidelines as closely as possible. To help the programmer maintain consistency of the user interface the company provides object classes for all the major controls, windows, and menus the user sees. Programmers are inherently lazy; they will use NeXT's interface objects, and therefore create applications that conform to the user interface guidelines, rather than create their own slightly "better" interface that confuses a user jumping from application to application. It's likely that they will also create applications more quickly.

The layout of the graphical interface portion of the program is usually done with NeXT's Interface Builder in a manner that isn't unlike the building of a simulation model in Ents. The programmer can drag over menus, windows, data entry forms, and buttons from a palette and drop them into his application. The objects are immediately instantiated, and connections to the programmer's own objects can be made by simply dragging out connecting lines.

For example, suppose we've written a Doorbell object that performs some useful internal function in our program, and we need some way to send messages to the object from the user. On the NeXT, we'd drag over a Button object and drop it into the application, then draw a line between the Button object and an icon that represents the internal Doorbell object. This sets the target of the button to be the Doorbell object; whenever the user clicks on the Button icon on the screen a message will be sent to the Doorbell, which can take whatever action it deems appropriate. Once the code is compiled the application is completely functional; the layout is not a prototype, but the actual, working program.
An Application class provides much of the most critical code, including a main event loop, utilities for managing the windows associated with the application, and connections to *Services*, an interapplication communication utility.

V.3. The simulation core.

The on-screen graphics manipulate an underlying model that does the actual work of the program. In this case the program decomposed itself into objects fairly readily: the individual nodes are each objects, and a simulation executive maintains the clock, the event list, and the random number generators.

The simulation core node objects are sent messages by the on-screen icon when connections among nodes are dragged out on the screen. Double-clicking on the icon for the node brings up an inspector panel which sends more detailed messages that control the internal state of the nodes.

Much of the code for a generic simulation core node can be put into an abstract class. Specific types of nodes—create nodes, activity nodes, and so on—are subclassed from an abstract Node class.

Entities are also objects. They are instantiated by Create nodes and kept in a central list by the simulation executive; they “move” through the system by the manipulation of pointers rather than the wholesale copying of data areas.

V.4. Dynamic allocation of memory.

The C language allows the programmer to dynamically allocate memory from the heap region at runtime. This is an extremely valuable capability; space to hold data or program objects can be claimed and freed on the fly.

A language without dynamic memory allocation must specify in advance the memory resources that the program will need. For example, in FORTRAN a popular way to keep track of entities is to use an array. The size of the array must be known when the program is compiled, so the user has to guess the number of entities that will be present. If the guess is too low the program will not function
correctly; the array would be filled and no more entities could be allocated. If the
guess is too high, memory will be unnecessarily allocated; perhaps only a few slots
of the array will be used by entities and the remainder will go to waste. In a
modern, multi-tasking, virtual memory environment this will impose a speed
penalty on other applications as well as the simulation program. The applications
share main memory resources, and one program’s excessive use will cause the
others to go without.

Dynamic memory allocation grants only exactly what memory is needed to
hold the data at any time. However, on the NeXT management (allocation and
deallocation) of the memory is the responsibility of the programmer. There are no
so-called “automatic garbage collection” utilities such as those in the Smalltalk or
LISP languages.

V.5. Speed.

No particular care was taken to make the program speedy. Great savings
could be made in some areas of the program, particularly the dragging operations in
the user interface and in some internal simulation engine aspects. The screen
updates that provide user feedback while the simulation is running—animation and
simulation clock display updates—consume substantial portions of the processor’s
time. In spite of this the program seems relatively responsive to the user in most
instances.

Objective-C is a compiled language and therefore executes faster than an
interpreted language. The object-oriented extensions to Objective-C consume a
small but very acceptable amount of overhead—most estimates put the cost of
invoking a method rather than a function at about 100% to 200%. Once in a
method or function the code will execute at the same speed as conventional C
programs. In an era of cheap, high speed hardware, a fixation on relatively minor
coding efficiencies at the expense of programmer productivity is misplaced.

V.6. Hardware and software requirements.

Ents will run on any NeXT machine running Operating System Release 2.0
or later. It will run on all hardware currently sold by NeXT.
V.7. Output.

The NeXT uses a single, unified graphics model for both screen displays and hardcopy output—PostScript. The PostScript language is a very popular and flexible high end page description language that has been extended to computer displays in addition to its traditional role as a page description language. The use of PostScript in both screen display and printer output immensely simplifies the programmers' task and increases the fidelity between what the user sees on the screen and what is obtained from the printer. Once the code that displays the graphics the user sees on the screen has been written, the same code can be used to display the picture on all PostScript devices at whatever resolution.

V.8. Verification

As a partial verification of the program's correctness, a simple M/M/1 queue was simulated and the results compared to those predicted by classical queuing theory. The analysis is presented in Appendix A.
VI. Conclusions

VI.1. Achievements.

The objective of this effort was to create a NeXT Step discrete event simulation tool that was easy to use and able to model a range of typical business systems. This goal has been met.

The program used Visual Interactive Simulation techniques to increase the feeling that the user was directly manipulating a physical model rather than working through an intermediary programming language, and restrictions on the ordering of nodes and the grammar of model creation were kept to a minimum. The standard conventions for NeXT applications were followed to decrease user learning time.

Object-oriented techniques were used to write code quickly and to reuse existing code. Reuse of the code was particularly high for the user interface objects defined by NeXT in the AppKit programming toolbox.

The program also provides a platform for further work on the distribution of the simulation and modeling process over several cooperating, communicating applications. It should not take much work to enable simulation data to be sent to graphing or analysis programs, for example.

VI.2 Limitations

As mentioned previously, the program does not support continuous or mixed discrete-continuous simulation.

Since layout of the blocks is currently non-hierarchical, scaling up to very large projects with thousands of nodes would be difficult; as it stands, the program is best used for small to perhaps medium sized projects. Beyond that point the so-called “surface area” of the model would become unwieldy, just as a program written in a single module is more difficult to understand than one broken up into a number of functions or procedures.
VI.3. Future enhancements.

Medieval cathedrals were built over a period of decades. The basic plan was sometimes altered while work was in progress, but building continued as time, money, and circumstances allowed. In the same way, most large programs are never really “finished.” Construction is always in progress. The high priests invariably want a new window design, prefer different icons, try to hold to their articles of faith in the face of temptation, and have few relics hidden away in the cellar.

Multiple execution threads would allow a simulation to run in the background while the user works on another model in another window. The simulations currently run in a modal loop; no other window within the application can be made the active window while the simulation is running, and the modal window is ordered at the top of the global window list. This is inconvenient and contrary to what most users expect. Threads would allow the user to interact with several running simulations at once. With remote object messaging and a networked cluster of machines, threads might also allow a sort of coarse-grained parallelism: several simulations running on different computers in the network, with the output displayed on the user’s local screen.

Entities are somewhat mysterious things in the current implementation—abstractions that are not made concrete and visible. The user is never given a feeling that the entities are being directly manipulated or even that they really exist. Allowing the user to drag entities to nodes or to directly edit the attribute/value list would reduce this feeling. The restriction on attribute value types is unnecessary as well. There is no reason that alphanumeric string values cannot be used as attribute values.

The interprocess communication facilities could be greatly improved. For example, it should be possible to click a button on a queue node inspector, have another application start up, and plot the length of a queue between two arbitrary points in time. Likewise, it should be possible to import node performance data into other programs, including statistics programs.
Analysis of data for multiple runs could be greatly simplified. Summary statistics on the result of each run could be collected and displayed in the node inspectors in much the same way individual run data is displayed now.

Combined discrete event and continuous simulation would be a pleasant addition for many people. The Mach operating system's use of threads could create some interesting capabilities. "Daemons" (processes that asynchronously monitor system states) could watch a continuous process and initiate an action when certain conditions have been met.
Bibliography


Sanderson, D. P., "CPSS: A C++ Library for Process-Oriented Simulation," unpublished manuscript, Department of Computer Science, University of Pittsburgh, PA 15260.


APPENDIX
Verification of M/M/1 Queue Performance

As a partial verification of the correctness of program logic, the output of a simple M/M/1 queue created in Ent was compared to the theoretical mathematical results predicted by queuing theory. The results showed no significant statistical basis for questioning the program's output.

The model selected was a minimal M/M/1 queue with FIFO service, an arrival rate of one per minute, and a service rate of one every 0.75 minutes. This implies an $\rho$ of 0.75.

Taha (1982) presents equations that predict the steady-state performance of the queue for several factors, including the average number of entities in the system, the average number of entities in line waiting for service, the average time spent in the system, and the average time spent in the queue.

To test the program's conformance to the theoretical results, 500 replications of 1000 minutes each were run on a simple single server queue model. At the end of each 1000 minute block the average values of the output parameters were examined, the queue’s statistical measures were reset to zero, and then execution resumed. The initial 1000 minute block was discarded to eliminate warm-up bias, leaving a total sample size of 499. There is a very small correlation between blocks, since the number of entities in line at the end of one block’s time period will carry over into the next; however, it was assumed that the length of the line and service time in relation to the block’s execution time would make this factor negligible.

Under Central Limit Theorem assumptions the output parameters will be normally distributed, and samples from the distribution will be distributed as a Student’s $t$ with $n-1$ degrees of freedom. For sufficiently large $n$ (greater than approximately 120) this approaches the Normal distribution. The test statistic

$$(X - \mu)/(S/(n)^{1/2})$$

will, if sufficiently large or small, indicate whether the hypothesis
H0: \( X = \mu \) vs. H1: \( X \neq \mu \)

is a reasonable one.

A queue with the parameters specified above would have expected results as shown below:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Mean</th>
<th>Observed</th>
<th>sdev</th>
<th>t-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_s )</td>
<td>3.0</td>
<td>3.019</td>
<td>0.7151</td>
<td>0.58</td>
</tr>
<tr>
<td>( L_q )</td>
<td>2.25</td>
<td>2.265</td>
<td>0.6898</td>
<td>0.50</td>
</tr>
<tr>
<td>( W_s )</td>
<td>3.0</td>
<td>3.003</td>
<td>0.6591</td>
<td>0.11</td>
</tr>
<tr>
<td>( W_q )</td>
<td>2.25</td>
<td>2.252</td>
<td>0.6458</td>
<td>0.05</td>
</tr>
</tbody>
</table>

In this case none of the t-values exceed a reasonable critical value, eg 1.64 for a two-sided, \( \alpha = 0.1 \) test while using the Normal approximation to the Student's t distribution.

No amount of testing will show the absence of all bugs. However, in at least this respect the program appears to work as expected.