DEVELOPMENT OF A C-BASED SIMULATION TOOLKIT
SUPPORTING DISCRETE, CONTINUOUS, AND COMBINED SIMULATION

by

Fazal U. Khan

Thesis submitted to the faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Master of Science
in
Industrial and Systems Engineering

APPROVED:

Michael P. Deisenroth, Chairman

Subhash Sarin
Dr. S. C. Sarin

Dr. O. K. Eyada

January 1991
DEVELOPMENT OF A C-BASED SIMULATION TOOLKIT
SUPPORTING DISCRETE, CONTINUOUS, AND COMBINED SIMULATION

by

Fazal U. Khan

Committee Chairman: Dr. M. P. Deisenroth
Industrial and Systems Engineering

(ABSTRACT)

In this research, a C-Based Simulation Toolkit (CBST) was developed. It supports discrete, continuous, and combined simulation.

CBST is a group of simulation support functions written in the language C. CBST functions are used within a specific framework similar to that of GASP IV. It employs the event scheduling world view for next event selection. The Runge-Kutta-Fehlberg integration method is used to update state variables.

The simulation framework of CBST, descriptions of CBST functions, and details of program construction using CBST are described. Four models are developed using CBST and the results are analyzed.

CBST is compared to similar software packages, namely GASP IV, DISC, and CSIM. CBST has more flexibility than GASP IV because it is C-based and because it offers resource
management functions. CBST supports continuous and combined simulation, whereas DISC and CSIM do not.
Acknowledgements

This body of work is dedicated to the memory of my grandparents.

My deepest thanks go to my family. I hope that I never forget how fortunate I am to have a family like mine. To my parents, Zafar and Zahida; my brothers, Asad and Sami; and my sister, Shehla; thanks for your love and support throughout this process. I could never have completed this work without your faith and encouragement. Also, thanks to my parents for funding most of this research.

For their friendship and support, I thank Bill Epling, Jim Potter, Wayne Weaver, Naheed Qureshi, Darryl Duncan, and Nitin Advani. Thanks to Anissa, Seema, and Aisha for interesting phone conversations. To my cousins in Lahore, Pakistan; thanks for the letters. Special thanks to my former roommates Ziaullah Khan, Vipul Doshi, and Amine Alameddine for their help.

To Dr. M. P. Deisenroth ("Dr. D"), my advisor, thanks for the most interesting topic and thanks for your quick reading times. Thanks also to Dr. Eyada and Dr. Sarin for serving on my committee.
Table of Contents

1. Introduction .................................................. 1
   1.1 Problem Statement ........................................ 2
   1.2 Objective .................................................. 4

2. Literature Review ............................................ 6
   2.1 Approaches to Simulation ................................. 6
      2.1.1 Discrete Simulation ................................. 7
      2.1.2 Continuous Simulation .............................. 10
   2.2 Simulation Software ..................................... 11
   2.3 The C Programming Language ............................. 15

3. Methodology .................................................. 19
   3.1 Program Development ..................................... 19
   3.2 Model Testing ............................................. 21

4. Basic Simulation ............................................... 23
   4.1 Definitions and Conventions ............................... 23
   4.2 Simulation Using the Definitions and Conventions ....... 26
      4.2.1 Discrete Simulation ................................. 26
      4.2.2 Continuous Simulation .............................. 29
      4.2.3 Combined Simulation ................................. 32

5. The C-Based Simulation Toolkit ............................... 36
   5.1 CBST Variables and Structures ........................... 36
      5.1.1 Discrete Simulation .................................. 36
      5.1.2 Continuous Simulation ............................... 43
      5.1.3 Combined Simulation ................................. 45
   5.2 CBST Functions ............................................ 45
5.2.1 Executive Control ...................... 46
5.2.2 CBST Variable Initialization .......... 52
5.2.3 List and Entity Management ............ 53
   5.2.3.1 Event List Management .......... 53
   5.2.3.2 List Management ............. 55
   5.2.3.3 Entity Management ........ 56
5.2.4 Resource Management .................. 57
5.2.5 Random Variate Generation ............. 58
5.2.6 Data Collection, Statistical
   Computations, and Reporting .......... 61
5.2.7 Error Handling ...................... 63
5.2.8 State Variable Management .......... 64

6. Using CBST .................................. 68
   6.1 Declarations ........................... 68
   6.2 User Written Functions ............... 69
      6.2.1 Discrete Simulation ............ 71
      6.2.2 Continuous Simulation .......... 72
      6.2.3 Combined Simulation .......... 75
   6.3 Compiling a Program .................. 75

7. Models Using CBST .......................... 76
   7.1 M/M/1 Queueing System Model ........... 76
      7.1.1 Objectives of the Model .......... 76
      7.1.2 Description of the System ........ 76
      7.1.3 The CBST Model ................. 78
      7.1.4 The Results .................... 79
   7.2 Cedar Bog Lake Example ............... 79
List of Figures

Figure 2.1  Software approaches to simulation ............ 12
Figure 4.1  Descriptive flowchart of
discrete simulation in CBST ............... 28
Figure 4.2  Descriptive flowchart of
continuous simulation in CBST ............ 31
Figure 4.3  Descriptive flowchart of
combined simulation in CBST .............. 33
Figure 5.1  The relationship of discrete() with user
written functions and CBST functions ... 47
Figure 5.2  The relationship of continuous() with user
written functions and CBST functions ... 48
Figure 5.3  The relationship of combined() with user
written functions and CBST functions ... 49
Figure 6.1  The relationship of advance() with user
written functions and CBST functions ... 73
Figure 7.1  M/M/1 queueing system ..................... 77
Figure 7.2  Expected behavior of tank/truck model ...... 85
Figure 7.3  Layout of the CIM Lab ..................... 87
Figure 7.4  Layout of the CIM Lab
with numbered locations ................... 91
Figure 7.5  Possible moves for a pallet
between numbered locations ................ 92
Figure 7.6  Model specifications of CIM Lab program .... 93
1. Introduction

Simulation is a powerful analytical tool that is widely used in industry. Simulation, in this context, refers to the process of designing a "mathematical-logical model" [21] of a system on a computer, using the computer to evaluate the system states numerically through time while gathering data, and using the data to estimate some of the true characteristics of the system. [9]

In manufacturing, it is often desirable to learn something about the behavior of a system whether in the stage of planning, implementing, or modifying. Sometimes, an analytical solution to a question may exist. Most of the time, however, manufacturing systems are much too complex to evaluate mathematically without making many simplifying assumptions. To model a complex system more realistically, simulation is often used.

Using simulation, the analyst can observe the resulting changes throughout a system based upon modifications made at one or several stages. Analytical methods might help to foretell the effect of a change in a limited fashion, but simulation can aid in predicting the effect upon the whole manufacturing system. For example, adding a machine to a system may cause the bottlenecks to occur at different stages. [9]
Simulation allows the user to estimate the behavior of a real system in an off-line environment. Simulation can help in the planning of a proposed system. On-line testing of a presently running system can be avoided through the use of a simulation model. This is advantageous if the system is costly to run or hazardous when experimenting. It may be an objective to test the limits of certain stressful variables and off-line testing presents a more cost effective means. [21] In addition, simulation may save much time over on-line testing. It is possible to simulate weeks of production time in relatively little time on a computer.

The use of simulation in manufacturing analyses has been growing rapidly in the last few years. Some factors contributing to this increase are: complexities in automation systems, decreases in computing costs, greater sophistication in simulation software, and the integration of graphical animation in simulation packages. [10]

1.1 Problem Statement

There are four basic categories of software for simulation in manufacturing. These are: (1) manufacturing simulators, (2) general purpose simulation languages,
(3) general purpose simulation tools, and (4) general purpose programming languages (such as C, FORTRAN, or Pascal).

Manufacturing simulators only deal with a narrow class of problems. Thus, many analysts have to use one of the remaining three approaches. Approach 2 can cover a wide variety of problems, but sometimes a system is too complex for these languages to handle efficiently. Approaches 3 and 4 are used in this instance.

Using approach 4 will entail writing a lengthy program that will be extremely complex. This problem arises because there are several mechanisms needed for a simulation and the use of approach 4 would require the developing of most or all of them. In approach 2, these are hidden from the user. Approach 3 combines the flexibility of approach 4 and the pre-existing simulation aids of approach 2. Approach 3 tools usually consist of a group of subroutines that support discrete event simulation, continuous simulation, or a combination. Usually, these tools have a general purpose language as their base. A program written using approach 3 will have a mixture of statements of the base language and calls to simulation aiding subroutines.

C is a good language on which to base a discrete event simulation language. It is a language that is used by many
serious programmers and is gaining more use in industry. It is a powerful and flexible language that can handle the special data constructions that would be helpful in simulation.

Presently, there are two C based simulation language supersets of the approach 3 type. One offers the process view and the other offers the event scheduling view. Neither of the two have provisions for continuous or combined simulation. Continuous models are those in which the variables are affected by continuous system state changes, whereas in discrete event models, the variables of interest are those that may change because of instantaneous changes in state. In combined simulation, both types of variables and their interactions are of interest.

There is a need for a C-based toolkit that can handle discrete, continuous, and combined simulation and that can allow some of the strengths of the C language to be exploited.

1.2 Objective

This research was intended to develop a simulation toolkit, written in C, that helps in discrete, continuous, and combined simulation modeling. The research demonstrated
the usefulness of the toolkit by simulating various models including the Computer Integrated Manufacturing Laboratory (CIM Lab) in Whittemore Hall.

The major thrust of the research was to develop the routines needed to support discrete, continuous, and combined simulation. These include executive control, system variable initialization, list and entity management, resource management, random variate generation, data collection, statistical computation, reporting, error checking, and state variable management. The routines were developed in a modular fashion so that the toolkit can be easily extended later and can thus be used as a basis to develop further simulation software such as graphical simulation and interactive simulation.

Finally, a simulation model of the CIM Lab was developed using the software package. This showed the applicability of the toolkit in simulating a real system. During this stage, several features specific to this model were built. This demonstrated the extensibility of the simulation software.
2. Literature Review

This review covers three areas that pertain to the research undertaken. First, some approaches to simulation are discussed. Second, a brief overview of simulation software related to the software being developed is included. Third, the C programming language is reviewed.

2.1 Approaches to Simulation

Typical system simulation models in manufacturing (and in many other applications) are usually classified as discrete change, continuous change, or combined. The words discrete and continuous in this case refer to dependent variables of interest. Generally, in a simulation study, the major independent variable is time. [21]

In discrete simulation, the dependent variables undergo instantaneous discrete changes at certain specified points in simulation time. These points are called event times. Time, an independent variable, may also be classified as continuous or discrete. If the time variable is continuous, the discrete changes in the dependent variables can occur at any point in time. If discrete, the changes in the dependent variables occur only at certain points. [21]
In continuous simulation, the dependent variables are allowed to vary continuously through simulation time. As in discrete simulation, the time variable may be continuous or discrete. If the time variable is continuous, the values of the dependent variables can be updated at any point in simulation time. If it is discrete, only at certain times are the dependent variables updated. [21]

In combined simulation, some dependent variables change discretely, some change continuously, and some may change continuously with instantaneous discrete changes at some points. Again, the independent time variable may be continuous or discrete. The interplay between the discretely changing variables and the continuously changing variables is usually the major concern of these simulations. [21]

2.1.1 Discrete Simulation

Discrete simulation is typically performed using the following conventions. The objects of interest within the system are called entities. These can be people, workstations, raw material, workpieces in progress, and the like. In a complex simulation there are many different types of entities coexisting and interacting. Each entity has certain traits associated with itself. These are termed
attributes. Sometimes, certain entities may have to be grouped together, perhaps because of a similar attribute. These groupings are called files or sets. Entities in the same file have some similarity with each other. [21]

Discrete simulation is an attempt to numerically mimic the activities of the entities and to make inferences about the behavior of the real system based on the outcome. Activities that move the system between states are identified. The attributes of the entities are given values. These are used to specify the state of a system. The system is in a specific state if the attributes of the entities are at levels which are consistent with the definition of that state. In discrete simulation, the system is moved from state to state through time in a manner analogous to the functioning of the real system. [21]

The system can only move from one state to another at event times. There are two types of events. The unconditional event is ready to be executed when it is scheduled, depending only on the time variable. Conditional events might depend on time and other conditions such as whether a resource is busy or idle. [6] A complete representation of the system can be acquired by advancing the simulation time from one event time to the following event time since the state of the system does not change.
between event times. This method of advancing is called the next event approach. Most discrete simulation languages follow it. [21]

There are three commonly used strategies that simulation languages employ for next event selection. These three are known as event scheduling, activity scanning, and process interaction. The strategy chosen affects how the modeler has to view the system and thus the three are sometimes called the world views. Most simulation languages use one of the three strategies and this is sometimes used to classify them. Some simulation languages, such as SLAM II, SIMSCRIPT II.5, and SIMAN offer more than one world view to the user. [6]

In the event scheduling method, there is a series of unconditional events through simulation time. Event notices are placed in an event list. The event notice with the earliest time of occurrence is selected. Then the simulation clock is updated to that time and the code for the particular event is executed. In many cases, part of the event processing code involves scheduling the next event of that particular type. The selecting, scheduling, and processing are repeated until termination of the simulation. [6]
Using the event scheduling approach entails much work by the programmer rather than the simulation executive of the simulation software. The activity scanning and process interaction approaches involve less control by the programmer. As a result, the event scheduling approach is the most flexible of the three. It can be used in a wide range of problems and is a good choice when building special purpose simulations for projects. [6]

2.1.2 Continuous Simulation

In continuous simulation, the system is represented by a group of equations which may be algebraic, difference, or differential. They usually describe time dependent variables and the simulation involves observing the system status while time is advanced. [21]

Many continuous languages use the Runge-Kutta methods for integration to update the sets of equations involved. The process is documented in numerous texts. It has two strengths which make it a good choice to use in a continuous simulation package. It is simple to adjust the step size. This is important since events might not be spaced uniformly. Also, Runge-Kutta integration is self-starting. Therefore, after an event, there is no problem in restarting. [21]
2.2 Simulation Software

Due to the differing needs in industry, there are several types of tools available for simulation in manufacturing. Depending on the problem, one could use (1) manufacturing simulators, (2) general purpose simulation languages, (3) general purpose simulation languages which are a superset of simulation supporting subroutines and some general purpose programming language (such as FORTRAN, C, or Pascal), or (4) general purpose programming languages. The four approaches are listed in logical order, with the first being the most sophisticated, easiest to use for the layman, least costly in model development time, and least flexible. Examples of the four software approaches are listed below in Figure 2.1.

Some examples of approach 1 packages are: FACTOR, HEI RTSS, InterFaSE, MAST, MIC-SIM, Micro SAINT, PROMOD, SIMFACTORY, STARCELL, WITNESS, and XCELL+. [10] These are frequently used for high level, aggregate analyses where the control logic for the system is not modeled, such as in the initial design stage of a manufacturing system. [10] Since manufacturing simulators only deal with a small class of problems, many analysts have to use one of the remaining three approaches.
### Approach 1: Manufacturing Simulators
- FACTOR, HEI RTSS, InterfaSE, MAST, MIC-SIM, Micro SAINT, PROMOD, SINFACCTOR, STARCELL, WITNESS, XCELL+

### Approach 2: General Purpose Simulation Languages
- AutoMod II, CADmotion, GPSS/PC, INSIGHT, PChModel, RESQ, SIMAN/Cinema, SIMPLE_1, SIMSCRIPT II.5, SLAM II, SLAMSYSTEM

### Approach 3: General Purpose Simulation Toolkits
- C based: DISC, CSIM
- Pascal based: SIMTOOLS, SIMPAS, PASSIM, INTERACTIVE, PASION
- FORTRAN based: GASP IV, SIMNET, CODESIM, SIMLIB
- Ada based: FAST, ASSE, SAMOA, A-SIM
- Modula-2 based: SIMOD
- PROLOG based: T-Prolog
- BASIC based: MicroSim
- PL/I based: SIMPL/I

### Approach 4: General Purpose Programming Languages
- C, C++, Pascal, FORTRAN,
- Ada, Modula-2, PROLOG,
- BASIC, PL/I, others

---

**Figure 2.1 Software approaches to simulation**
Using approach 2 is still fairly simple and a wide variety of problems can be handled. Examples of simulation languages are: AutoMod II, CADmotion, GPSS/PC, INSIGHT, PCModel, RESQ, SIMAN/Cinema, SIMPLE_1, SIMSCRIPT II.5, SIAM II, and SLAMSYSTEM. [10] Occasionally, the need arises to model an overly complex model that approach 2 cannot handle easily. In this case, approach 3 or 4 must be used.

Much of the simulation analysis in industry is performed using approach 4. However, when this approach is used, the required code can become too large and too complex to be feasible for most systems. It has been stated that among programs of this type, a minimum of 90% of the coding is general purpose and a maximum of 10% is specific to the model at hand. [34] This is because several mechanisms common to most simulations are needed and must be programmed. In approach 2, these are part of the language and are hidden from the user. Using approach 4 would entail writing a large amount of code for the mechanisms and for coordinating them.

Approach 3 languages provide a middle ground between approaches 2 and 4. They usually consist of a group of subroutines that support discrete event simulation. A few of these support continuous and combined simulation as well. A typical program written using one of these languages
consists of a mixture of the base language (FORTRAN, C, Pascal, etc.) and calls to the routines as needed. This option has the flexibility of lower level tools with some of the simulation aids of higher level tools.

Numerous simulation tools of the approach 3 type have been developed using several different general purpose programming languages. Examples are: DISC [30] and CSIM [27] (C based); SIMTOOLS [29], SIMPAS [3], PASSIM [35], INTERACTIVE [17], and PASHON [22] (Pascal based); GASP IV [20], SIMNET [32], CODESIM [23], and SIMLIB [11] (FORTRAN based); FAST [25], ASSE [1], SAMOA [13], and A*SIM [14] (Ada based); SIMOD [12] (Modula-2 based); T-Prolog [2] (PROLOG based); MicroSim [31] (BASIC based); and SIMPL/I [15] (PL/I based).

Three of the approach 3 tools are of particular interest to this research because they are the most similar to CBST. These are GASP IV, DISC, and CSIM.

GASP IV was developed by Pritsker. [20] Its base language is FORTRAN. The event scheduling approach is used for discrete simulation. GASP IV provides support for continuous and combined simulation as well. The framework used in CBST was based on the general framework of GASP IV.
DISC was developed by Selvaraj, et. al. [30] It uses C as the base language. It employs the event scheduling approach for discrete simulation and does not provide support for continuous or combined simulation. DISC provides several interaction options to the user.

CSIM was developed by Schwetman. [27] The base language is C. It provides support for the process view. Continuous and combined simulation are not supported in CSIM. CSIM has some features which make it helpful in simulating the operation of computer systems and system components and has been used frequently for this purpose.

2.3 The C Programming Language

C is currently one of the most extensively used programming languages in the world. It was originally developed by Dennis Ritchie in 1972 from Ken Thompson’s language, B, as an aid to designing the Unix system. The Unix system and a majority of the programs that run on it are written in C. It was designed as a language to aid in systems programming, which entails writing code for operating systems, text editors, compilers, interpreters, and the like. C, however, is being used in a number of other applications such as text processing programs, database systems, telephone-switching equipment, engineering
programs, numerical analysis, and hardware control. Further evidence of the popularity of C is the fact that C compilers are available for most computers. [8]

C has a number of virtues as a programming language. It is compact, yet powerful. It allows low level abilities such as addressing of memory. It is easily portable compared to other languages. Also, it encourages modular design and can be extended.

C is compact in that it has relatively few keywords. For example, it has 30 keywords and 13 statement types while Modula-2 has 40 keywords and 10 statement types. However, C is a terse and powerful language and thus, programs written in it are typically shorter than programs written in Modula-2. [18]

It derives its power from its control structures and the various data types that it allows. These control structures can be built upon each other to produce many levels of control. In addition, C has several basic data types which can be combined in an almost unlimited number of ways to produce new data types suitable for the programming task at hand. This produces a great deal of flexibility to handle a variety of applications. [18]

Through the use of "pointers," C allows indirect addressing of memory. In this sense, C is similar to
assembly languages and thus is sometimes called a mid-level language. Pointers can be used for several purposes in C. Sometimes, the address of a variable is passed to a function so that it can be manipulated directly without having to make a copy. This is especially useful when dealing with an array or other multi-valued data type. The use of pointers can be combined with the use of a data type called a structure to produce various data structures such as linked lists and binary trees. C has several operators for manipulating pointers and the use of pointers is one of the most powerful aspects of the language.

C is considered to be an easily portable language. This means that a C program written on one computer can be used on a different computer with little modification. This property is derived from the compactness of C and the fact that it was originally defined using the PDP-11, which is a "small machine." [18] Usually, all that is needed to port a program from one computer to another is a few changes in the header files. [37]

During the past few years, the portability of C has been put in question due to a number of different compilers using different conventions on a few aspects. However, a standard version of C, called ANSI C, has been decided upon
and this will bring about fewer problems in using the same program with different compilers and different computers.

The design of C forces one to employ good programming habits such as top-down planning, structured programming, and modularity. [37] This is especially useful in large programs where it would be hard to keep track of all variables and how a change in the program would affect each. In C programming, local variables are used whenever possible. Thus, the effect of a change would be localized and modifications would become much easier.

C is an "extensible" [18] language in that other languages can be developed using C as a subset. Examples of this are C++, Concurrent C [18], and CSIM [27]. In addition, C has several standard libraries of often used functions. [18]

C is a powerful language that has many strong points. It is a popular programming language in a number of fields. It has some weaknesses, although with some skillful programming, most of these can be overcome. [33]
3. Methodology

The purpose of this research was to develop a simulation support toolkit based in the general purpose programming language, C. The toolkit supports discrete, continuous, and combined simulation. In addition, the toolkit was designed utilizing many of the advantages of C such as modularity, portability, and flexibility. A key objective was to design the toolkit such that it can be easily extended.

3.1 Program Development

The development of the basic functions required for discrete, continuous, and combined simulation was the main phase of the research. The tools for discrete event simulation were implemented first. Next, continuous and combined simulation procedures were developed. Finally, tracing ability was added to the software. The software package is called the C-Based Simulation Toolkit (CBST). The programming was done using the C programming language. A Sun 386i workstation was used as the environment for programming.

CBST provides support in the form of functions and a general framework for discrete (event scheduling), continuous, and combined simulation. These include the
following classes: (1) executive control, (2) CBST variable initialization, (3) list and entity management, (4) resource management, (5) random variate generation, (6) data collection, statistical computation, and reporting, (7) error handling, and (8) state variable management.

Separate simulation executive functions were developed for discrete, continuous, and combined simulation. These are called from main() to control the flow of the program. The executives utilize the basic core functions listed above.

Functions were made to initialize CBST variables such as simulation times, statistics for the event lists, and statistics for lists and resources.

Functions were written to manage the event list, other lists, and entities. The event list and the other lists are doubly linked lists. An event notice can be placed on the event list, and the first event notice can be taken off the event list. An entity can be placed on a list in some priority order (FIFO, LIFO, lowest value first, highest value first), and the first entity on a list can be removed. Memory space in the heap is allocated for an entity dynamically through a CBST function.
Through calls to CBST functions, the modeler can check the availability of a resource, capture units of a resource, release units of a resource, increase the units in a resource, or decrease the units of a resource.

CBST random variate generation functions include a random number generator and functions to generate random deviates from them. The following distributions are supported: exponential, uniform, Erlang, Weibull, beta, gamma, normal, lognormal, Bernoulli, binomial, and Poisson.

Various CBST functions collect data, perform statistical computations, and report them. Two types of variables are of special interest: those that are based on observations, and those that are time persistent.

There are some functions to trap errors and report them. Three functions are used to terminate the running of a program and provide an error message.

CBST procedures for handling state variables and for supporting continuous and combined simulation were developed.

3.2 Model Testing

Four models were developed using CBST to demonstrate its use and its usefulness. The first three were simple
examples of discrete, continuous, and combined simulation. The fourth was a model of a more complex system -- the Computer Integrated Manufacturing Laboratory (CIM Lab) in Whittemore Hall at Virginia Polytechnic Institute and State University.
4. Basic Simulation

4.1 Definitions and Conventions

The definitions and conventions followed by the C-Based Simulation Toolkit (CBST) in regard to discrete, continuous, and combined simulation affect how the user models a system under study. These are discussed below.

As in GASP IV, the main concern is "modeling a system in two dimensions: the time dimension and the state-space dimension." [20]

The time dimension can be broken down into three elements: outlining of the conditions that cause each event, definition of possible changes to the system for each event, and sequencing of the events. The user is responsible for the first two elements. CBST provides routines for the third element, thereby obviating the need for the user to order the events.

The state-space dimension includes entities, their attributes, resources, and state variables. Entities are the model representations of items that flow through the system. Examples are: customers requiring services, workpieces requiring machining operations, or transportation vehicles unloading materials into the system. The attributes are characteristics of the entities such as:
specific services required by a customer, the size of a workpiece, or the amounts and types of materials carried by a truck. These are sometimes termed "discrete attributes" because their values do not change between event times. Resources are typically items that stay fixed within the system and usually provide some type of service to the entities, for example: bank tellers, machining stations, or unloading cranes. State variables are involved in continuous and combined simulation. These are sometimes called continuous, or dynamic variables because their values can change between event times. Examples of these include the level of a liquid in a tank, the path of an automated guided vehicle, or the wear of a tool.

The definition of an event in CBST is the same as in GASP IV:

"An event occurs at any point in time beyond which the status of a system cannot be projected with certainty." [20]

This implies that although events usually effect a change in the status of a system, they need not in all cases. In addition, applying the definition above, change in the system status can occur continuously without the occurrence of an event. [20]

Events can be classified as either time-events or state-events. Time-events are those that are scheduled to
occur at a particular point in time. They are associated with next event simulation. State-events transpire when state variables meet user defined conditions rather than being scheduled for a specific point in time. A state-event can cause a time-event to be scheduled and a time-event can cause a state-event to occur. [20]

Events bring about changes to the system in three ways. State variables and/or attributes of entities can be changed at event times. The interrelationships between entities or state variables can be altered. Finally, the entities present in the system can be changed by adding new ones, removing old ones, or both. Typically, the logic involved with a time-event includes the scheduling of one or more events, one of which is often of the same type. [20]

Another important concept in the use of CBST is that of lists (sets, files, queues). A list is a collection of entities that have some relationship to each other. Lists are most often used to hold entities waiting for a specific resource. In these cases they are sometimes called queues or files. Lists can also be used to collect entities that have finished service, that have balked out of a queue, or other such occurrences. In these cases they are sometimes called sets.
The schedule of time-events is kept on a particular list called the event list. The event list is a list of event notices kept in chronological order. Each event notice contains the event type, its time of occurrence, and a reference to the entity involved with the event (a pointer to an entity structure in the case of CBST). When events are scheduled, they are placed on the event list according to their time of occurrence. When the next event is required, it is removed from the head of the list.

4.2 Simulation Using the Definitions and Conventions

Given the definitions and conventions above; discrete, continuous, and combined simulation are implemented in specific ways.

4.2.1 Discrete Simulation

Discrete simulation involves entities, their attributes, resources, time-events, lists, and the event list. State variables and state-events are not involved.

Entities are created when needed. They are assigned attributes when it is deemed necessary by the modeler. They are placed on the event list when they are involved with a future event such as arrival into the system or completion
of a service by a resource. Entities may also be placed in lists (or sets). This may occur if an entity is waiting in a list for the availability of a particular resource or if an entity is placed in a set with some specific characteristics such as finished products.

Pending their availability, units of resources are captured and released by entities. When a resource is captured, the entity involved is placed on the event list along with the type of event (in this case, the end of service by the resource) and the time of occurrence (the time of the ending of the service). When the event occurs, the unit(s) of resource are released and the entity is taken off the event list. A decision is then made by the modeler to place the entity back onto the event list (if it is involved with another future event), to place the entity into a set or list, or to destroy the entity.

A descriptive flowchart of how discrete simulation is carried out in CBST is shown in Figure 4.1. The CBST function `discrete()` handles this task.

First, CBST and user defined variables are initialized. Next, a loop is entered from which the only way to exit is to meet a stopping condition. Stopping conditions are defined by the user. Examples of stopping conditions are: the simulation time has exceeded a specific value, the
Figure 4.1 Descriptive flowchart of discrete simulation in CBST
number of entities that have gone through the system surpasses a value, and the number of entities waiting for a specific resource reaches some value. After the stopping conditions are checked, the next event is removed from the event list. This is the event notice that has the earliest time of occurrence and is therefore located at the head of the event list. The simulation time, or tnow, is then advanced by setting it equal to the time of occurrence of the removed event notice. The time-event logic associated with the type of event of the removed event notice is then carried out. Control then loops back to the stopping conditions check and the process is repeated until a condition is met.

4.2.2 Continuous Simulation

Continuous simulation involves state variables and state-events. Entities, their attributes, resources, time-events, lists, and the event list are not involved.

State variables are assigned initial values by the user. The way they change through simulation time is defined by the user in the form of differential, difference, and/or algebraic equations. The user may define conditions for a state-event to occur. In this case, the state-event logic codes must also be written by the user.
A descriptive flowchart of how continuous simulation is carried out in CBST is shown in Figure 4.2. The CBST function `continuous()` handles this task.

First, CBST variables, user defined variables, and local status flags are initialized. Next, the local variable representing the time at the beginning of a step, \( \text{tbeg} \), is set equal to \( \text{tnow} \), a global variable representing the simulation clock time. Stopping conditions are then checked. These may be conditions such as: a certain time has been reached or a state variable has exceeded a specific value. Next, \( \text{tend} \), a local variable representing the time to which \( \text{tnow} \) will be advanced if no state-events occur, is calculated. \( \text{tend} \) is set equal to the smaller value of either the time of the end of the next step, or the next time to record state variables. The size of each step and the increments at which to record state variable values for plotting are determined by the user. Flags are set based upon which value is smaller. If the values are close to each other (their difference is less than \( \text{treq} \), a value set equal to 1/100 of the maximum step size), then flags are set to both advance to the end of the next step and record state variable values for plotting.

An attempt is then made to advance \( \text{tnow} \) from \( \text{tbeg} \) to \( \text{tend} \). If \( \text{tnow} \) is advanced to \( \text{tend} \), then no state-events
Figure 4.2  Descriptive flowchart of continuous simulation in CBST
occurred in the interval. Based on the flag values, the state variable values may be recorded, and/or the flags may be reset. Control is then sent back to the previous step where \( t_{beg} \) is set equal to \( t_{now} \). If \( t_{now} \) is not advanced to \( t_{end} \), then a state-event occurred. In this case, the state-event logic is carried out, \( t_{now} \) is set to the time of the state-event, and control is sent back to the previous step where \( t_{beg} \) is set equal to \( t_{now} \).

4.2.3 Combined Simulation

Combined simulation involves all aspects of discrete and continuous simulation. In a sense, discrete and continuous simulation are special cases of combined simulation. In combined simulation, entities, their attributes, resources, time-events, lists, the event list, state-events, and state variables are involved.

A descriptive flowchart of how combined simulation is handled in CBST is shown in Figure 4.3. The CBST function \texttt{combined()} handles this task.

First, CBST variables, user defined variables, and the local status flags are initialized. Next, the local variable representing the time at the beginning of a step, \( t_{beg} \), is set equal to \( t_{now} \), a global variable representing
Figure 4.3 Descriptive flowchart of combined simulation in CBST
the simulation clock time. Stopping conditions are then checked. These may be conditions such as: a certain time has been reached, a state variable has exceeded a specific value, or a specific number of entities have passed through the system. Next, \texttt{tend}, a local variable representing the time to which \texttt{tnow} will be advanced if no time-events occur, is calculated. \texttt{tend} is set equal to the smallest value of: the time of occurrence of the next time-event; the time of the end of the next step; or the next time to record state variables. The event list is checked to find the time of the next time-event. The size of each step and the increments at which to record state variable values for plotting are specified by the user. Flags are set based upon which value is smallest. If the values are close to each other (their difference is less than \texttt{tres}, a value set equal to 1/100 of the maximum step size), then flags are set to perform two or all three of the operations.

An attempt is then made to advance \texttt{tnow} from \texttt{tbeg} to \texttt{tend}. If \texttt{tnow} is advanced to \texttt{tend}, then no state-events occurred in the interval. Based on the flag values, time-event logic may be carried out, the state variable values may be recorded, and/or the flags may be reset. Control is then sent back to the previous step where \texttt{tbeg} is set equal to \texttt{tnow}.
If \( t_{\text{now}} \) is not advanced to \( t_{\text{end}} \), then a state-event occurred. In this case, the state-event logic is carried out, \( t_{\text{now}} \) is set to the time of the state-event, and control is sent back to the previous step where \( t_{\text{beg}} \) is set equal to \( t_{\text{now}} \).
5. The C-Based Simulation Toolkit

Some specific variables and structures that are employed in the C-Based Simulation Toolkit (CBST) are discussed below with regard to their C implementations. In addition, the CBST functions are described.

5.1 CBST Variables and Structures

5.1.1 Discrete Simulation

The simulation clock time is kept using a global variable called \texttt{tnow}. \texttt{tnow} is initialized to 0.0 by the CBST function \texttt{discpinit()}; it may be reset to another value by the user via \texttt{unit()}. \texttt{tnow} is referred to by many functions and is advanced by several functions.

The information needed to manage a resource is kept in a structure of type \texttt{resnote}. The declaration of this structure is included in CBST and is shown below.

```c
struct resnote {
    char *name;    /* name tag */
    int cap;       /* capacity */
    int utilcur;   /* current util */
    tpvar util;    /* utility stats. */
    double busy;   /* busy time counter */
    boolean bflag; /* busy flag */
    double idle;   /* idle time counter */
    double idltimmax; /* max idle time */
    double bsytimmax; /* max busy time */
    long int entcount; /* entity count */
};
typedef struct resnote resnote;
resnote rn[NRES];
```
name is a pointer to a character array indicating the name of the resource. This may be declared by the modeler if needed. cap is the capacity of the resource in number of units. This must be declared by the modeler in unit().
utilcur is the number of units of the resource that are currently captured. This is set equal to 0 by default at the beginning of program execution but can be changed by the modeler. util is a structure of type tpvar which is the record of utilization of the resource, a time persistent variable. bflag is a boolean variable indicating whether all units of the resource are captured or not. busy is the current amount of time that all units of the resource have been captured. bsytimmax is the maximum amount of time that all units of the resource have been captured. idle is the current amount of time that all units of the resource have been idle. idltimmax is the maximum amount of time that all units of the resource have been idle. entcount is the number of entities that have used the resource. All of the variables in resnote are initialized by setting them equal to zero. It is the modeler’s duty to change cap and any other variable that is necessary.

There is one structure of type resnote for each resource. A global array of NRES structures of type resnote is created by CBST. The array is called r[n[]]. The number
of resources in the model must be specified at the beginning of the program by a \texttt{#define} statement. To specify that there are 6 resources in a model, the following statement would be used:

\texttt{\#define NRES 6}

It should be noted that \texttt{NRES} must be defined for CBST to execute discrete simulation. The minimum value that \texttt{NRES} can have is 1.

To set the capacity of the last resource equal to 2 units in a model with 6 resources (\texttt{NRES = 6}), the following statement would be used:

\texttt{rn[5].cap = 2;}

It should be remembered that array index subscripts start with 0 in C.

The information needed to manage a list is kept in a structure of type \texttt{listnote}. The declaration of this structure is included in CBST and is shown below.

\begin{verbatim}
struct listnote {
    char *name; /* name of list */
    int lmax;  /* max length */
    int lcur;  /* current length */
    tpvar length; /* length stats. */
    obsvar wait; /* wait stats. */
};

typedef struct listnote listnote;
listnote evb;
listnote lb[NLISTS];
\end{verbatim}
name is a pointer to a character string that denotes the name of the list. lmax is the maximum length of the list thus far. lcur is the current length of the list. length is a time persistent variable concerning the length of the list. wait is a variable based on observation dealing with the times that entities spend in the list. All of the variables in each listnote are initialized to zero by default.

There is one structure of type listnote for the event list and one for each list. The listnote structure for the event list is called evb. A global array of NLISTS structures of type listnote is created by CBST. The array is called lb[]. The number of lists in the model must be specified at the beginning of the program by a \#define statement. To specify that there are 5 user defined lists in a model, the following statement would be used:

\#define NLISTS 5

It should be noted that NLISTS must be defined for CBST to execute discrete simulation. The minimum value that NLISTS can have is 1. To refer to the length of the last list, in the case when NLISTS = 5, lb[4].lcur would be used.

Two types of variables are of interest in a simulation model: time persistent variables, and variables based on observation.
The declaration for a structure of type `tpvar` (dealing with a time persistent variable) is shown below.

```c
struct tpvar {
    double ttot;    /* current time */
    double tlast;   /* time of last update */
    double tclr;    /* time of last clear */
    double mean;    /* mean of tp var. */
    double areah;   /* accumulated area */
    double nharea;  /* accum. area of h^2 */
};
typedef struct tpvar tpvar;
```

This type of variable is used in `listnotes` and `resnotes`. In addition, the modeler may declare variables of type `tpvar`. The function `accumarea()` operates on variables on this type. The inputs to `accumarea()` are a double, a pointer to a `tpvar`, and another double. The first double is the value of the variable at current simulation time. `accumarea()` should be called every time the value changes. The second double is the time at which the change occurred. `accumarea()` sets `tlast` to `ttot`. `ttot` is reset to the time at which the variable changed, the third input. The quantity of the current value of the variable times the difference between `ttot` and `tlast` is added to `areah`. The quantity of the square of the current value of the variable times the difference between `ttot` and `tlast` is added to `nharea`. `tclr` represents the last time that the statistics for the variable were cleared. It is set to 0 by default and is changed to the current simulation time every time the function `clrtp()` is called. `accumarea()` also calculates the
average of the variable each time that it is called and stores this in \textbf{mean}.

The declaration for a structure of type \texttt{obsvar} (dealing with a variable based on observation) is shown below.

\begin{verbatim}
struct obsvar
    { double num;    /* number of obs. */
      double mean;  /* mean of variable */
      double sumx;  /* sum of observations */
      double sumxx; /* sum of obs^2 */
    };

typedef struct obsvar obsvar;
\end{verbatim}

This type of variable is used in \texttt{listnotes} and may also be used by the modeler. The function \texttt{notevalue()} operates on variables of type \texttt{obsvar}. The inputs to \texttt{notevalue()} are a double representing the value of the variable and a pointer to the structure of type \texttt{obsvar} in question. \texttt{notevalue()} increments \texttt{num} by 1.0 every time that it is called. The double is added to \texttt{sumx} and the square of the double is added to \texttt{sumxx}. The average of the variable is calculated and this is placed in \textbf{mean}.

The event list is implemented as a doubly linked list. Event notices on the event list are kept in structures of type \texttt{element}. The declaration is shown below.

\begin{verbatim}
typedef struct element element;
struct element
    { element *before;
      element *after;
      int evtype;
      double evtime;
      double tentry;
      entity *pent;
\end{verbatim}
});
element *evlist = NULL;

before is a pointer to the element immediately before and
after is a pointer to the element immediately after. evtype
is the type of event. evtime denotes the time of event.
pent is a pointer to the entity involved in the event. Note
that instead of holding a complete entity, only a reference
to the memory location of the entity is carried, thus
obviating the need to move the entity's in memory
unnecessarily.

Functions schedule() and nextevent() operate on the
event list. schedule() is used by the modeler to place
event notices onto the event list. nextevent() is used by
CBST to remove the event with the earliest time of event --
the next event. Both schedule() and nextevent() update
statistics concerning the event list.

The user should specify the number of random number
streams that will be used by the model. This should be done
at the beginning of the program by a #define statement. To
declare that there are 4 random number streams in a model,
the following statement would be used:

#define NSEEDS 4

It should be noted that NSEEDS must be defined for CBST to
execute discrete simulation. The minimum value that NSEEDS
can have is 1. When using more than 1 seed, the user must
initialize the seeds by giving values to each in the global array called `seed[]`. The number must be a negative integer with a value between -1 and the largest negative number allowed for an integer in the implementation of C being used. To set the first seed to -123, the following statement would be used:

```c
seed[0] = -123;
```

This may be done in `main()` or in `uinit()`.

### 5.1.2 Continuous Simulation

As in discrete simulation, the simulation clock time is kept using a global variable called `tnow`. `tnow` is initialized to 0.0 by the CBST function `contpinit();` it may be reset to another value by the user via `uinit()`. `tnow` is referred to by many functions and is advanced by the CBST function `advance()`.

In continuous simulation, the user must specify 2 global constants via `#define` statements. These are `NODE`, which represents the number of state variables, and `NSTEVF`, which represents the number of state-event flags. Both must be defined and the minimum value is 1.
A global array called sflag[] is used to help in monitoring whether a state-event is occurring and to find out which one is occurring. It contains NSTEVP integers.

In continuous simulation, the user must decide the step size, tdfull, and the interval between recording state variable values, tdrec. Both are global variables. Both should be set by the user in uinit(). In addition, the following global variables should be set in uinit(): hmin, hmax, and tol. hmin is the smallest step size that desolve() can take. It must be less than tdfull. A good range of values is tdfull/100 to tdfull/10. hmax is the largest step size that desolve() can take. It must be no larger than tdfull and must be greater than hmin. A good range is tdfull/2 to tdfull. tol is the tolerance value used in gamcal(). A good value for tol is tdfull/1000.

The global arrays snow[], slast[], dnow[], and dlast[], are used by CBST to keep track of state variable values and the rates at which they are changing. Each array holds NODE doubles. snow[] holds the current values of the state variables and slast[] contains the values at the previous step. Likewise, dnow[] holds the current rates at which the state variables are changing and dlast[] has the rates at the previous step.
5.1.3 Combined Simulation

Combined simulation involves all of the data types and data structures involved in discrete and continuous simulation.

5.2 CBST Functions

CBST provides support for many capabilities needed in discrete simulation (using event scheduling), continuous simulation, and combined simulation. These include:

(1) executive control, (2) CBST variable initialization, (3) list and entity management, (4) resource management, (5) random variate generation, (6) data collection, statistical computation, and reporting, (7) error handling, and (8) state variable management.

Discrete simulation requires the use of 1-7. Continuous simulation requires 1, 2, 6, 7, 8, and possibly 5. Combined simulation requires all support mechanisms.

The functions are listed in Appendix A. Descriptions of their use follow. A good working knowledge of the C programming language is assumed.
5.2.1 Executive Control

Executive control is provided by three functions: discrete(), continuous(), and combined(). One of the three is called by main() depending on whether the user wants to implement discrete, continuous, or combined simulation. Descriptive flowcharts of the respective algorithms are shown in Figures 4.1, 4.2, and 4.3. The relationships of discrete(), continuous(), and combined() with user written functions and CBST functions are shown in Figures 5.1, 5.2, and 5.3. The functions are listed in Appendix A.

In discrete(), the CBST variable initializing function discpinit() is called first. Then the user written initializing function uinit() is called. After this, an indefinite loop is entered. The user written function to check for stopping conditions, stopcheck(), is called. If it returns a positive non-zero number, then program flow is sent back to main(). The event list is then checked to make sure that there is a notice on the list. The next event notice is then taken off the list using nextevent() and the pointer to the element is assigned to pel. tnow is set equal to the time of occurrence of the event. then the user written function tevents() is called and pel is sent to it. After tevents() is through, the memory space in the heap occupied by the removed event notice is freed for later use. The loop ends here. The only ways to get out of the loop
Figure 5.1 The relationship of discrete() with user written functions and CBST functions
Figure 5.2 The relationship of continuous() with user written functions and CBST functions
Figure 5.3 The relationship of combined() with user written functions and CBST functions
are to meet a stopping condition or to run into some kind of error which aborts program execution.

In continuous(), the CBST variable initializing function contpinit() is called first. Then the user written, initializing function uinit() is called. Flags and times are then initialized. The function bldarr() is then called to record the state variable values at the initial simulation clock time, 0 by default. After this, an indefinite loop is entered. The variable tbegin is set equal to tnow. The user written function to check for stopping conditions, stopcheck(), is called. If it returns a positive non-zero number, then program flow is sent back to main(). Next, the variable tend is set equal to the smaller of the time of the end of next step or the next time to record state variable values. The flags, ff2 and ff3, are set depending on which is smaller. Both flags may be set equal to 1 if the difference is less than tres. Next, the function advance() is called and the variables tbegin and tend are sent to it. advance() will move the simulation clock forward starting from tbegin while updating state variable values. It will check for state-events and find the first one in the interval if one exists. It will advance the simulation clock time up to tend. Program execution is now back in continuous(). If tnow was not advanced to tend, the flags are reset to 0 and the loop starts over. If tnow was
advanced to `tend` and the flag for recording state variable values for plotting is set, then `bdarr()` is called and then the loop starts over. Otherwise, the loop starts over without calling `bdarr()` or resetting the flags.

In `combined()`, the CBST variable initializing function `combpinit()` is called first. Then the user written, initializing function `uinit()` is called. Flags and times are then initialized. The function `bdarr()` is then called to record the state variable values at the initial simulation clock time, 0 by default. After this, an indefinite loop is entered. The variable `tbeg` is set equal to `tnow`. The user written function to check for stopping conditions, `stopcheck()`, is called. If it returns a positive non-zero number, then program flow is sent back to `main()`. Next, the variable `tend` is set equal to the smallest of: the time of the next state-event, the time of the end of next step, or the next time to record state variable values. The flags, `ff1`, `ff2`, and `ff3`, are set depending on which is smallest. More than one flag may be set equal to 1 if the difference is less than `tres`. Next, the function `advance()` is called and the variables `tbeg` and `tend` are sent to it. `advance()` will move the simulation clock forward starting from `tbeg` while updating state variable values. It will check for state-events and find the first one in the interval if one exists. It will
advance the simulation clock time up to tend. Program execution is now back in combined(). If tnow was not advanced to tend, the flags are reset to 0 and the loop starts over. If tnow was advanced to tend, then bldarr() is called if flag ff3 is positive and time-event logic is carried out if flag ff1 is positive. The loop then starts over. Otherwise, the loop starts over without calling bldarr() or resetting the flags.

5.2.2 CBST Variable Initialization

CBST provides system variable initialization by the functions discpinit(), contpinit(), combpinit(), and initlist(). Functions discpinit(), contpinit(), and combpinit() initialize global variables and functions for discrete, continuous, and combined simulation respectively. All three initialize one or more of the following to default values if needed: all global time variables, the structure for the event list, the structures that contain intermediate statistics for all resources and all lists, the state event flags, and entity counters.

Function initlist() is used by discpinit() and combpinit() to help initialize the structures of type listnote.
5.2.3 List and Entity Management

This class of routines can be divided into three areas: (1) event list management, (2) set (list) management, and (3) entity management.

5.2.3.1 Event List Management

Functions for event list management include schedule(), nextevent(), cancell(), cancellall(), fixevlist(), insert(), and makeelement(). Function schedule() is used by the modeler in discrete and combined models. It calls the lower level function insert(), which in turn calls makeelement(). Function nextevent() is used by the executive functions discrete() and combined() to take the next event off the event list. Functions cancell() and cancellall() are used to take event notices of a specific type off the event list. Both use fixevlist() to rearrange pointers in the event list.

Function schedule() takes as input an integer representing the event type, a double representing the time of the event occurrence, and a pointer to the entity involved in the event. schedule() calls insert() to place an event notice on the event list. schedule() then updates statistics dealing with the event list. Finally, schedule() returns a pointer to the head event notice.
The input for `cancel1()` and `cancelall()` is an integer representing the event type that is to be removed from the event list. Both functions return a 1 if an event of this type is found. Otherwise the return is a 0. `cancel1()` takes off the first event notice of the specific type. `cancelall()` takes off all event notices of the specific type.

The inputs for `insert()` are a pointer to the head `element` of a list, an integer, two doubles, and a pointer to an `entity`. When `insert()` is used by `schedule()`, the pointer to `element` is a pointer to the head of the event list, the integer is the event type, the first double is the time of event, the second double is the current simulation clock time (`tnow`), and the `entity` pointer points to the `entity` involved with the event. `insert()` finds the correct place on the event list based upon the event time. It then calls `makeelement()`, which allocates space in the heap for a structure of type `element` and returns a pointer to the `element`. `insert()` then assigns the values to the proper places in the `element` including pointers to the `element` to its left and to its right on the event list. Finally, `insert()` returns a pointer to the head of the list. `insert()` and `makeelement()` are modified versions of C functions of the same name developed by Van Wyk. [36]
5.2.3.2 List Management

CBST functions for list management include the high level functions filefifo(), filelifo(), filelvf(), filehvf(), removelst(), removenum(), and removeSPEC(). Also in this group are the low level functions insfifo(), inslifo(), insprior(), insert(), and fixlist().

Functions filefifo() and filelifo() take as input an integer representing the number of the list, and a pointer to the entity that is to be placed on that list. These functions first call insfifo() or inslifo(). These functions insert the entity pointer onto the list in a first in, first out or a last in, first out manner. Functions filefifo() and filelifo() then call pfisestat(), which updates statistics concerning that list. These statistics are kept in a structure of type listnote.

The inputs for filelvf() and filehvf() are an integer representing the list number, a pointer to the entity entering the list, and an integer representing the furthest level of ranking attributes to check. filelvf() places an entity pointer onto a list in a lowest value first manner. filehvf() is used for highest value first. Both functions use the r[] array of the entity to rank them. The user must define the entity with an array called r[] to use these functions. Both functions call insprior() to place the
entity pointer onto the particular list. insprior() ranks the entity either lowest or highest value first, depending on which function called it. pfilestat() is used to update the file statistics.

Functions removelst(), removenum(), and removespec() are used to attempt to remove an entity pointer from a specific list. All three input the list number, an integer. removenum() also inputs another integer representing the rank of the entity pointer to be removed. removespec() also inputs a double and looks in r[0] of each entity in the list to find a match. All three return a NULL if the proper entity is not found and return a pointer to the entity if it is found. All three call mfilestat() to update statistics concerning the particular list. Each of the three use fixlist() to rearrange the pointers in the doubly linked list used to implement the list.

5.2.3.3 Entity Management

The only function solely for entity management is makeentity(), which allocates space in the heap for a structure of type entity and returns a pointer to the entity. entity pointers are also manipulated in the functions for event list management and list management.
5.2.4 Resource Management

Functions for resource management include `resavail()`, `rescapture()`, `resrelease()`, `resincr()`, and `resdecr()`.

The input for `resavail()` is an integer representing the resource index number. This function returns the number of units of a resource still available and generates a fatal error message if this number is negative. This function can be used by the modeler to see if units of a resource are available before capturing them or for some other logical decision.

The inputs for `rescapture()`, `resrelease()`, `resincr()`, and `resdecr()` are two integers. The first is the resource index number and the second is the number of units. `rescapture()` captures a number of units of the specific resource and updates the resources statistics kept on its structure of type `resnote`. `resrelease()` releases units of a resource and updates its statistics. `resincr()` and `resdecr()` increase or decrease the capacity of the specific resource by the number of units inputted to the respective function. `resdecr()` makes sure that at least that number of units are available before decreasing the capacity of the resource and generates a fatal error message if not.
5.2.5 Random Variate Generation

Random variate support is provided by the functions: ran3(), expon(), uniform(), erlang(), weibull(), beta(), beta1(), gamma(), gam1(), gam2(), normal(), gaussian(), lognormal(), triang(), triang1(), bernoulli(), binomial(), and poisson(). Functions ran3(), normal(), and gaussian() are based on C functions found in Press, et. al. [19] Functions expon(), erlang(), weibull(), beta(), beta1(), gamma(), gam1(), gam2(), lognormal(), triang(), triang1(), bernoulli(), binomial(), and poisson() are based on algorithms found in Law and Kelton. [12]

Support for some of these functions is provided by beta1(), gam1(), gam2(), gaussian(), and triang1(). It should be noted that ran3() is used in the generation of any random deviate. Furthermore, expon(), gamma(), normal(), and bernoulli() are called by some of the functions.

All of the random deviate functions have n as their first input. n is an integer representing the random number stream to be used. It ranges between 0 and (MSEEDS-1). Functions bernoulli(), binomial(), and poisson() return integers. All of the others return doubles.

ran3() returns a sample from a uniform(0,1) distribution. It uses the subtractive method of generating random numbers. It uses random number stream n.
The other input to \texttt{expon()} is a double, \texttt{beta}, denoting the mean of the distribution. \texttt{expon()} returns a sample from an exponential distribution with a mean of \texttt{beta}.

The other inputs to \texttt{uniform()} are 2 doubles, \texttt{lo} and \texttt{hi}, representing the beginning and the end of the interval. \texttt{uniform()} returns a sample from a uniform[\texttt{lo}, \texttt{hi}] distribution.

The other inputs to \texttt{erlang()} are a double, \texttt{beta}, and an integer, \texttt{m}. \texttt{beta} is the mean and \texttt{m} is the number of times the exponential distribution is called. It returns a sample from an \texttt{m}-Erlang distribution with a mean of \texttt{beta}.

The other inputs to \texttt{weibull()} are \texttt{alpha} and \texttt{beta}. Both are doubles. \texttt{weibull()} returns a sample from a Weibull distribution with shape parameter \texttt{alpha} and scale parameter \texttt{beta}.

The other inputs to \texttt{beta()} are 4 doubles: \texttt{a}, \texttt{b}, \texttt{alpha1}, and \texttt{alpha2}. \texttt{beta()} returns a sample from a beta distribution with shape parameters \texttt{alpha1} and \texttt{alpha2} on the interval [\texttt{a}, \texttt{b}].

The other inputs to \texttt{gamma()} are \texttt{alpha} and \texttt{beta}; both are doubles. \texttt{gamma()} returns a sample from a gamma
distribution with shape parameter \( \alpha \) and scale parameter \( \beta \).

The other inputs to \( \text{normal()} \) are \( \text{mean} \) and \( \text{stdev} \), 2 doubles. \( \text{normal()} \) returns a sample from a normal distribution with mean \( \text{mean} \) and standard deviation \( \text{stdev} \).

The other inputs to \( \text{lognormal()} \) are \( \text{mean} \) and \( \text{stdev} \), both doubles. \( \text{lognormal()} \) returns a sample from a lognormal distribution by using the inputs to generate a normal random variate, \( z \), and returning \( y = \exp(z) \).

The other inputs to \( \text{triang()} \) are \( \alpha, \beta, \) and \( \gamma \). All 3 are doubles. \( \text{triang()} \) returns a sample from a triangular distribution on the interval \([a, b]\) with mode \( \gamma \).

The other input to \( \text{bernoulli()} \) is \( p \), the chance of success. \( p \) should be between 0.0 and 1.0. \( \text{bernoulli()} \) returns a 1 if the trial was a success, and 0 for a failure.

The other inputs for \( \text{binomial()} \) are \( t \), an integer, and \( p \), a double. \( t \) is the number of trials to conduct. \( p \) is the chance of success in each trial. \( \text{binomial()} \) returns an integer between 0 and \( p \). The integer represents the number of successful trials.

The other input to \( \text{poisson()} \) is an integer, \( \text{mean} \), representing the expected value. \( \text{poisson()} \) returns an
integer which is a sample from a Poisson distribution with
an expected value of mean.

5.2.6 Data Collection, Statistical Computations, and
Reporting

Functions `notevalue()`, `accumarea()`, `clrov()`, `clrtp()`,
`pfilestat()`, `mfilestat()`, `crlist()`, `clrres()`, `rlist()`,
`rresource()`, `rlhead()`, `rrhead()`, `replist()`, `repres()`,
`bldarr()`, `getarray()`, and `cplot()` provide support in this
area.

`notevalue()` is used for statistics based on
observation. The inputs are a double, representing the
observation of a variable, and a pointer to the structure of
type `observ` which holds the statistics for that variable.
`notevalue()` updates the mean and some temporary statistics
for the specific variable.

`accumarea()` is used for time persistent variables. The
inputs are a double, representing the value of the variable,
a pointer to the structure of type `tpvar` which holds the
statistics for that variable, and a double indicating the
time that the variable is updated. `accumarea()` updates the
mean and some temporary statistics for the specific
variable.
clears the statistics for a variable based on observation. The input is the address of the structure of type obsvar that is to be cleared. clrt() clears the statistics for a time persistent variable. The input is the address of the structure of type tpvar that is to be cleared.

pf() is used by several functions which place an entity into a list (file) to update statistics dealing with that list, specifically those that are kept on a structure of type listnote. pf() is used by functions which take entities off lists. Both have as their input an integer representing the index number of the list.

clrs() and clres() are used to clear the statistics of lists and resources. clrs() clears all lists except the event list by modifying values in the listnotes of the lists. clres() clears all resources by modifying values in the resnotes. Statistics affected by clrs() are: maximum length of a file, mean and standard deviation of the length of a file, and mean and standard deviation of the waiting time in the file. Statistics affected by clres() are: mean and standard deviation of the utilization, maximum idle time, maximum busy time, and entity count.

rlist() prints current statistics for a specific list, rresource() for a specific resource. rlhead() prints a
heading for list statistics, rrhead() for resource statistics. replist() prints the time that it is called, a heading, and the current statistics for the event list and all lists. repres() does the same for all resources. The outputs of replist() and repres() can be seen in Appendix B.1a.

Functions bldarr(), getarray(), and cplot() are used to record state variable values at regular intervals and to plot them against time. bldarr() collects the values and places them in a two dimensional array. This array is dynamically allocated in the heap by using getarray(), which returns a pointer to an array of 50 arrays of doubles. getarray() is called initially by bldarr() and after every 50 calls to bldarr(). The inputs to bldarr() are a pointer to an array of doubles and a double. The array of doubles carries the state variable values and the double is the current time. cplot() is the function that generates a plot of the state variables against time. Its input is an integer. If the integer is 80, the output is formatted for 80 columns. Otherwise, the format is for 132 columns.

5.2.7 Error Handling

Aid in error handling is provided by the functions perror(), perror1(), perror2(), and checkt(). perror()
takes, as input, a pointer to a character array. This character array is usually a message indicating the type and location of the error. perror() displays the message, displays the current simulation time (tnow), and aborts execution of the program. perror1() is similar to perror(), except that its inputs are a pointer to a character array and an integer. The character string and the integer are used to display a message. As in perror(), tnow is then displayed and the program is aborted. perror2() is similar to the other two except that its inputs are two pointers to character arrays and an integer. As with the other two, these are used to produce an error message.

checkt() is used to make sure that certain variable values specified by the user make sense in relation to each other.

5.2.8 State Variable Management

CBST routines dealing with state variable management include cross(), advance(), desolve(), rkf(), gamcal(), update(), and whennext().

cross() is used by the modeler to test whether a threshold has been crossed by a state variable. It is used as an aid in finding out when state-events occur. The
inputs are 5 doubles and 1 integer. The doubles represent, respectively, the value of the crossing variable at the end of the last step, its value now, the value of the threshold variable at the end of the last step, its value now, and the tolerance allowed. The integer denotes the directions of crossing that is of interest. The integer is positive, negative, or zero depending on whether the modeler is looking for a positive crossing, a negative crossing, or a crossing in either direction. The function returns an integer. The return is 0, if there is no crossing; 1, if the crossing is within tolerance; and 2, if the crossing is not within tolerance.

The function advance() is one of the most important routines in CBST. It is used by continuous() and combined(). Its inputs are two doubles. The first, ta, is the current time and the second, tb, is the time to which the system is to be advanced. advance() attempts to advance the state variables and the time from ta to tb while watching for state-events. If there are one or more state-events, advance() will search for the first state-event within tolerance. An interval halving algorithm is used for this task. advance() sets the system time, updates the state variables, and calls the proper user written state-event function if necessary.
Functions `desolve()`, `rkf()`, and `gamcal()` form the differential equation solver that is used to update the state variables. They are adapted from FORTRAN subroutines in Johnson and Riess. [7] They are an implementation of the Runge-Kutta-Fehlberg (RKF) method based on a fourth- and fifth-order method.

`desolve()` has, as inputs, 2 doubles and 2 pointers to arrays of doubles. The first double, `a`, is the start time and the second, `b`, is the end time. The first array contains the values of the state variables at the start time and the second is an empty array in which the values of the state variables at end time are to be placed. Essentially, `desolve()` solves an initial value problem on the interval `[a,b]`. `desolve()` controls the choice of step size as it solves the problem across the interval. `desolve()` uses `rkf()` and `gamcal()` to implement the RKF method and to estimate the error. The step-size is adjusted such that the error is within user specified limits.

`update()` is used by `advance()`. It updates the global arrays `snow[]` and `dnow[]`, where `snow[]` holds the values of the state variables at the current simulation time, or `tnow`, and `dnow[]` has the rates of change for the state variables at `tnow`. 
whennext() is a simple function that returns a double indicating the time the next event is scheduled to occur. It is used by combined().
Chapter 6. Using CBST

6.1 Declarations

First, the user should \texttt{#include} the stdio.h and math.h headers.

Then, the appropriate constants should be specified via \texttt{#define} statements. \texttt{NLISTS} represents the number of lists. \texttt{NRES} is the number of resources. \texttt{NR} is the number of doubles in the array \texttt{r[]} that will be in the \texttt{entity} definition. \texttt{NSEEDS} is the number of random number streams. \texttt{NODE} is the number of state variables. \texttt{NSTEVF} is the number of state event flags. Discrete simulation requires \texttt{NLISTS}, \texttt{NRES}, and \texttt{NSEEDS} to be set. Continuous simulation requires \texttt{NSEEDS}, \texttt{NODE}, and \texttt{NSTEVF} to be set. Combined simulation requires all 6 to be set. The minimum that each should be set to is 1. All 6 are used to set the size of various arrays and it should be noted that in C, the first \texttt{element} in an array of size \texttt{n} has the index 0 and the last has the index (\texttt{n}-1).

If discrete or combined simulation is being performed, then a declaration must be made for a structure of type \texttt{entity}. The attributes are the variables within the \texttt{entity} definition. There are no restrictions on the names or the data types of the variables in the declaration except that an array of doubles called \texttt{r[NR]} must be specified. This
array is used by the functions filelvf(), filehvf(), and removespec().

The modeler is required to use a typedef statement to conform to CBST implementation. An example is given below.

```c
struct entity {
    char *type;
    double tentry;
    double r[2];
};
typedef struct entity entity;
```

In the entity declaration above, the modeler is planning on monitoring different types of entities with type and their time of entry into the system with tentry. The modeler is using r[] to hold 2 values for filelvf(), filehvf() in the case where it may be necessary to prioritize on two levels.

The user must `#include "xdisc.h"` for discrete simulation or "xcont.h" for continuous. Finally, the modeler must `#include "xcbst.h"`.

### 6.2 User Written Functions

To use CBST for simulation, the modeler must write some specific functions. In addition, these functions must be written in a specific manner. Simple examples of these can be found in the first three models in Chapter 7.
For every simulation model using CBST, the user must write the functions `main()`, `uinit()`, and `stopcheck()`.

In its simplest form, `main()` is used to start simulations by calling one of the 3 executive control routines. The modeler may also use `main()` to make multiple runs of a model by changing parameters within the function and making the appropriate number of calls to one of the executive control routines.

In `uinit()`, the modeler initializes variables to other than CBST default values and initializes non-CBST variables if needed. For example, the user would set the values of the capacities for any resources in this function. `uinit()` may also be used to schedule initial events onto the event list. `uinit()` is called by the each of the executive controllers. It does not return any values and thus is declared as a void function.

`stopcheck()` is used to check for stopping conditions in the simulation. Examples of conditions are: the simulation time has exceeded a certain value; the level of a state variable has been surpassed, or a specified number of entities have passed through the system. `stopcheck()` returns an integer and must be written such that it returns 0 if no stopping condition has been met and returns 1 or any
positive integer if a stopping condition is met.
*stopcheck()* is called by each of the executive controllers.

### 6.2.1. Discrete Simulation

The relationship of *discrete()* with user written functions and CBST functions is shown in Figure 5.1.

In discrete simulation models using CBST, the function *tevents()* must be written by the modeler. In addition, the modeler must write an event function for each time-event.

*tevents()* is employed to send control to the proper time-event functions. It must be written to input a pointer to an *element*. It has no return and is declared a *void* function. It should use the pointer to *element* to access the event type and use this information to decide which time-event function to call. It should send the pointer to *element* to this time-event function.

The time-event functions are where the modeler describes, in C code, what happens at the occurrence of each time-event. They should be written to input a pointer to *element*. This pointer to *element* may be used to access the pointer to the *entity* associated with it. The modeler may use the pointer to *entity* to place the entity in a list. In many cases, the next occurrence of that event is scheduled
as part of the event function. Time-event functions may also be used to report certain statistics at fixed times.

6.2.2. Continuous Simulation

The relationships of \texttt{continuous()} and \texttt{advance()} with user written functions and CBST functions are shown in Figures 5.2 and 6.1.

To implement continuous simulation using CBST, the modeler must write the functions \texttt{yfirst()}, \texttt{stevcond()}, \texttt{sevents()}, and any event functions needed.

\texttt{yfirst()} is the function in which the modeler describes the changes to the state variables, usually in the form of differential equations. It must be written to input a double, and two pointers to doubles. The double is the reference to the independent variable, time. The first pointer points to an array representing the values of the dependent variables. The second points to an array representing the rate of change of the dependent variables. As an example, the equation: \( \frac{dy}{dt} = 2ty \), could be coded as:

```c
void yfirst(t, y, dy_dt)
double t, *y, *dy_dt;
{
    dy_dt[0] = 2.0 * t * y[0];
}
```
Figure 6.1 The relationship of advance() with user written functions and CBST functions
The number 0 is used to reference the first double in both arrays. The names of the variables above do not have to be used, but the general structure of the function has to be similar. The arrays are used so that systems of equations can be accommodated.

Function `stevocond()` is written by the modeler to test for conditions indicating that a state-event has occurred. Typically, this involves testing whether a state variable has passed a threshold and setting a global array of integers, called `sflag[]`, accordingly. `sflag[]` is used by the executive control routine to find occurrences of state-events.

Function `sevents()` is called by the executive control routine. It should check the global vector `sflag[]` and call the proper state-event functions accordingly.

The state-event functions are where the modeler indicates what happens when a state-event occurs. Typically, the following changes may be made in a state-event function:

1. Changes in the equations for state variables;
2. A discrete change in the value of a state variable;
3. Changes in the state-event conditions;
4. Changes to CBST or non-CBST variables.
6.2.3. Combined Simulation

The relationships of `combined()` and `advance()` with user written functions and CBST functions are shown in Figures 5.3 and 6.1.

In combined simulation, using CBST, all of the functions for discrete simulation and continuous simulation are used. All of the functions are written in the same manner as given above.

In combined simulation, time-event and state-event functions have more options than in discrete or continuous simulation. For instance, in combined simulation a time-event function may look at the value of a state variable to make a decision, or may change the value of a state variable. A state-event function may schedule an event to occur or may put an entity into a list.

6.3 Compiling a Program

To compile the program in a UNIX environment, the user would type:

`cc progname.c -lm`

where `progname.c` is the name of the program.
7. **Models Using CBST**

To demonstrate the usefulness of the software package, four models are developed using CBST. The first three are simple systems. The first is a discrete simulation of an M/M/1 queueing system. The second is a continuous simulation model example called Cedar Bog Lake. [21] The third is a combined simulation model. [24] The fourth is a model of a more complex system: the Computer Integrated Manufacturing Laboratory (CIM Lab) in Whittemore Hall.

7.1 **M/M/1 Queueing System Model**

7.1.1 **Objectives of the Model**

The objectives of this model are to present a simple example of how to use the package for discrete simulation and to observe some of the output values in relation to the analytical solutions of those values.

7.1.2 **Description of the System**

Figure 7.1 shows the system. It consists of a server whose service time is exponentially distributed with a mean of \( x \) minutes, and a queue to hold waiting customers. Customers arrive into the system with an interarrival time that is exponentially distributed with a mean of \( y \) minutes.
Customers arriving with an interarrival time that is exponentially distributed with a mean of \( y \) minutes

Server whose service time is exponentially distributed with a mean of \( x \) minutes

Figure 7.1 M/M/1 queueing system
The server can serve only one customer at a time. Customers that arrive into the system and find the server busy must wait in the queue and are handled in a first come, first serve manner.

In this analysis, the mean interarrival time is 10 minutes; that is, \( y \) is equal to 10. Three different values are used for \( x \), the mean service time. These are 9, 7, and 5. According to the analytical solutions; at steady state the average delay in queue should be 81, 16.33, and 5 minutes respectively. Similarly, the utilization of the server should be 0.9, 0.7, and 0.5 respectively.

7.1.3 The CBST Model

The server is modeled as a resource (resource 0) with a capacity of 1, the customers as entities which are created when they enter the system and destroyed when they leave the system, and the queue as a list (list 0) that is handled in a first in, first out (FIFO) manner.

Using the main() function in Appendix B.1b, 10 runs of each model were executed. The random number seed was changed for each of the 10 runs. main() was written to make a customized report using some of the CBST functions. Using
the method of independent replications, the results of the 10 runs of each of the 3 models were used to construct 95% confidence intervals.

A listing of a program to handle a simple case of an M/M/1 queueing system is contained in Appendix B.1a. The rewritten function main() that handles all three models is shown in Appendix B.1b. This is a demonstration of the flexibility of CBST.

7.1.4 The Results

The output of the simple model is shown in Appendix C.1a. Appendix C.1b has the output using the rewritten main() function. The expected values of average delay in queue and average utilization fall into the 95% confidence intervals in all 3 cases.

7.2 Cedar Bog Lake Model

7.2.1 Objectives of the Model

The objectives of this model are to demonstrate the use of CBST in a simple continuous simulation model and to compare the output with the output of SLAM II.
7.2.2 Description of the System

The model for Cedar Bog Lake is described by Pritsker, 1986. [21] The following are the variables involved:

\[ y[0] = \text{energy content of plants} \]
\[ y[1] = \text{energy content of herbivores} \]
\[ y[2] = \text{energy content of carnivores} \]
\[ y[3] = \text{amount of organic sediment} \]
\[ y[4] = \text{energy losses to the environment} \]
\[ y[5] = \text{energy gains due to solar radiation} \]

The cycle of solar radiation is given by the algebraic equation:

\[ y[5] = 95.9 \times (1 + 0.635 \times \sin (2\pi t)). \]

The variables are related to time and to each other through the following differential equations:

\[ \frac{dy[0]}{dt} = y[5] - 4.03y[0], \]
\[ \frac{dy[1]}{dt} = 0.48y[0] - 17.81y[1], \]
\[ \frac{dy[2]}{dt} = 4.85y[1] - 4.65y[2], \]
\[ \frac{dy[3]}{dt} = 2.55y[0] + 6.12y[1] + 1.95y[2], \text{ and} \]
\[ \frac{dy[4]}{dt} = 1.00y[0] + 6.90y[1] + 2.70y[2]. \]

The initial values of these variables are: \( y[0] = 0.83, \)
\( y[1] = 0.003, y[2] = 0.0001, \) \( y[3] = 0.0, \) \( y[4] = 0.0. \)
7.2.3 The CBST Model

The program for the Cedar Bog Lake example using CBST is shown in Appendix B.2. Below is an example of how the function `yfirst()` could be written for this example:

```c
void yfirst(t, y, ddy)
{ double t, *y, *ddy;
  double temp;
  temp = 95.9 * (1.0 + 0.635 * sin(2.0 * PI * t) );
  ddy[0] = temp - 4.03 * y[0];
  ddy[1] = 0.48 * y[0] - 17.87 * y[1];
  ddy[4] = y[0] + 6.9 * y[1] + 2.7 * y[2];
}
```

It should be noted that the variable for solar radiation is represented by the local variable `temp` and is therefore not plotted in the output.

7.2.4 The Results

The CBST results are shown in Appendix C.2a. The output from SLAM II is shown in Appendix C.2b. The plots of the state variables are comparable with the exception of the solar radiation variable.
7.3 Tank/Truck System Model

7.3.1 Objectives of the Model

The objectives of the model are to demonstrate the use of CBST in a simple combined simulation model and to compare the results with expected results. Of special interest is the interaction of the discrete and continuous elements of the model.

7.3.2 Description of the System

This is a model of a liquid waste storage system. The system includes a tank and an unloading bay. The tank has a capacity of 12,000 gallons. When the level of the tank goes above 10,000 gallons, a valve is opened to allow some of the liquid to leave the tank at a constant rate. The valve then remains open until the level of the tank goes below 7,000 gallons. The tank empties at a rate of 200 gallons per minute when the valve is open. When the level goes above 9,500 gallons, the loading bay is closed and remains so until the level goes below 9,500 gallons.

Trucks enter the system with 2000 gallons of liquid waste. If the loading bay is open and idle, they empty it into the tank and then leave the system. The unloading bay can accommodate only one truck at a time. Trucks that arrive
when the unloading bay is busy must wait in a first in, first out (FIFO) queue. The truck at the head of the queue may later use the unloading bay when it becomes idle, if it is open (liquid level below 9,500 gallons.) The trucks empty their contents at the rate of 200 gallons per minute.

The initial level of the tank is 6,500 gallons. The loading bay is initially open. The valve is closed. There are no trucks in the system. The first truck enters the system at time = 5 minutes, the second at time = 12.5 minutes, and the third at time = 14 minutes.

7.3.3 The CBST Model

The trucks are modeled as entities. The level of the liquid in the tank is a state variable. The loading bay is modeled as resource 0, with a capacity of 1. The open/closed statuses of the loading bay and the valve are modeled by the flags: fbay and fvalve.

The time-events are: (1) the arrival of a truck to the system, and (2) end of service for a truck. The state-events depend on the level of the liquid. They are: (1) crossing of the 9,500 gallon level in a positive direction (close the bay), (2) crossing of the 10,000 gallon level in a negative direction (open the valve), (3) crossing
of the 9,500 gallon level in a negative direction (open the bay), and (4) crossing of the 7,000 gallon level in a negative direction (close the valve).

7.3.4 The Results

The expected results are shown in Figure 7.2. The output from the program is in Appendix C.3. The output was custom made for this model using various CBST utility functions. The output matches that which is expected.

Figure 7.2 depicts the expected events that take place in this system. Truck 1 arrives at time = 5.0 minutes. The loading bay is open and is idle, so it starts service. Truck 2 arrives at time = 12.5. The loading bay is busy, so truck 2 waits in the queue. Truck 3 arrives at time = 14.0. The loading bay is busy, so the truck goes in the queue behind truck 2. Truck 1 finishes service at time = 15.0 and exits the system. The loading bay is still open and now it is idle. The queue is checked; truck 2 captures the bay and starts emptying. At time = 20.0, the level of the tank crosses 9500 gallons in a positive direction and thus the loading bay is closed. Truck 2 continues to empty into the tank. At time = 22.5, the level of the tank surpasses 10,000 gallons in a positive direction and the valve is opened as a result. Truck 2 finishes emptying at 25.0 and
Figure 7.2 Expected behavior of tank/truck model
leaves the system. Since the loading bay is still closed, truck 3 cannot start service. At time = 27.5, the level of the tank reaches 9500 gallons at a negative slope. The loading bay is opened as a result. The queue is checked and truck 3 starts service. Truck 3 finishes service at time = 37.5 and leaves the system. The valve is still open and the tank drains until time = 50.0, when the level of the tank reaches 7000 gallons coming at a negative slope. The valve is closed at this time. No new liquid is added or drained after this time.

7.4 CIM Lab Model

7.4.1 Objectives of the Model

The objective of this model is to demonstrate the use of CBST in a "real world" model. An important aspect of this model is the complexity of the scheduling and routing rules.

7.4.2 Description of the System

The layout of the CIM Lab is depicted in Figure 7.3. The CIM Lab consists of an assembly station, a machining station, an operator, an automatic storage/retrieval system, and a conveyor system to connect them. Two products are
Figure 7.3 Layout of the CIM Lab
made in the CIM Lab: miniature robot models and miniature CNC models. These are made from three types of raw materials: robot base blanks (wax), CNC base blanks (wax), and link blanks (wax).

The various parts are moved around the CIM Lab on five different types of pallets. These are:

1. CNC base blanks pallet: holds 9 CNC base blanks
2. Robot base blanks pallet: holds 12 robot base blanks
3. Link blanks pallet: holds 21 link blanks
4. CNC kit pallet: holds 1 CNC base blank and 1 link blank, or 1 finished miniature CNC model
5. Robot kit pallet: holds 1 robot base blank and 2 link blanks, or 1 finished miniature robot model

The assembly station consists of an IBM 7547 industrial robot, some fixtures to aid in assembly, and feeders for the three types of blanks. The CNC base, robot base, and link feeders hold 6, 6, and 11 blanks respectively.

There are seven possible operations at the assembly station. These are:

1. Restock CNC base blanks: move CNC base blanks from pallet type 1 to the CNC base blanks feeder (which has a capacity of 6 blanks).
2. Restock robot base blanks: move robot base blanks from pallet type 2 to the robot base blanks feeder (with a capacity of 6 blanks).
3. Restock link blanks: move link blanks from pallet type 3 to the link blanks feeder (with a capacity
of 11 blanks).

4. Build kit for CNC: move 1 CNC base blank and 1 link blank to an empty type 4 pallet.

5. Build kit for robot: move 1 robot base blank and 2 link blanks to an empty type 5 pallet.

6. Assemble CNC: move machined CNC base and machined link to assembly fixture from a type 4 pallet, assemble the two pieces into a finished miniature CNC model, and move this back to the type 4 pallet.

7. Assemble robot: move machined robot base and machined links to assembly fixture from a type 5 pallet, assemble the three pieces into a finished miniature robot model, and move this back to the type 5 pallet.

The machining station consists of an IBM 7545 industrial robot and 2 DYNAL CNC milling machines.

There are two operations that can take place at the machining station. These are:

1. Machine CNC: take 1 CNC base blank and 1 link blank from a type 4 pallet, perform milling operations on them, and return the machined parts to the type 4 pallet.

2. Machine robot: take 1 robot base blank and 2 link blanks from a type 5 pallet, perform milling operations on them, and return the machined parts to the type 5 pallet.

The operator is located beside the conveyor system. The operator removes finished products from type 4 and type 5 pallets, places raw materials onto pallets of type 1, 2, or 3, and handles any problems requiring human solution.
The automatic storage/retrieval system consists of a storage rack with 126 slots for storing the various pallets and a storage and retrieval machine which transports pallets from the storage rack to the conveyor system and vice versa.

Assuming that the feeders have adequate contents and excluding issues of scheduling and routing on the conveyors and AS/RS, the two products undergo the following steps:

1. Miniature CNC models: (a) assembly operation 4 on an empty type 4 pallet, (b) machining operation 1, and (c) assembly operation 6.

2. Miniature robot models: (a) assembly operation 5 on an empty type 5 pallet, (b) machining operation 5, and (c) assembly operation 7.

7.4.3 The CBST Model

The program for the CIM Lab model is in Appendix B.2. Figure 7.4 shows the CIM Lab with various numbered locations. These are possible positions where the pallets can stop. Position 0 is the storage rack. Positions 1 - 9 are on the conveyor system. Figure 7.5 shows the possible movements between the various locations.

Figure 7.6 shows the resources, lists, and event functions used in this model.

The even numbered resources represent the actual use of a specific resource, while the odd numbered resources are
The automatic storage/retrieval system consists of a storage rack with 126 slots for storing the various pallets and a storage and retrieval machine which transports pallets from the storage rack to the conveyor system and vice versa.

Assuming that the feeders have adequate contents and excluding issues of scheduling and routing on the conveyors and AS/RS, the two products undergo the following steps:

1. Miniature CNC models: (a) assembly operation 4 on an empty type 4 pallet, (b) machining operation 1, and (c) assembly operation 6.

2. Miniature robot models: (a) assembly operation 5 on an empty type 5 pallet, (b) machining operation 5, and (c) assembly operation 7.

7.4.3 The CBST Model

The program for the CIM Lab model is in Appendix B.2. Figure 7.4 shows the CIM Lab with various numbered locations. These are possible positions where the pallets can stop. Position 0 is the storage rack. Positions 1 - 9 are on the conveyor system. Figure 7.5 shows the possible movements between the various locations.

Figure 7.6 shows the resources, lists, and event functions used in this model.

The even numbered resources represent the actual use of a specific resource, while the odd numbered resources are
Figure 7.4 Layout of the CIM Lab with numbered locations
Figure 7.5 Possible moves for a pallet between numbered locations
Resources:
0  storage crane
1  storage crane (reserve)
2  assembly station
3  assembly station (reserve)
4  machining station
5  machining station (reserve)
6  conveyor
7  conveyor (reserve)
8  operator
9  operator (reserve)

Lists:
0  unfulfilled orders
1  fulfilled orders
2  waiting at assembly station
3  waiting at machining station
4  AS/RS for pallet type 1
5  AS/RS for pallet type 2
6  AS/RS for pallet type 3
7  AS/RS for pallet type 4
8  AS/RS for pallet type 5

Event functions:
0  order()
1  mv01()
2  mv14()
3  mv17()
4  mv20()
5  mv34()
6  mv43()
7  mv45()
8  mv47()
9  mv52()
10  mv54()
11  mv67()
12  mv76()
13  mv78()
14  mv82()
15  mv85()
16  mv89()
17  mv92()
18  mv95()
19  assbreak()
20  mbreak()
21  endmc()
22  endass()

Figure 7.6 Model specifications for CIM Lab program
used to reserve a particular resource so that it cannot be captured by an erroneous pallet.

List 0 holds unfulfilled orders, while list 1 holds fulfilled orders. List 2 is used to hold a pallet waiting at the assembly station for the conveyor. Likewise, list 3 is used to hold a pallet waiting at the machining station. The maximum number of pallets in the list should be 1. Lists 4 - 8 represent the AS/RS locations for pallets of type 1 - 5 respectively.

The event function \texttt{order()} simulates what happens at the arrival of each order. It also schedules the next order. The functions, 1 - 18, are the programming for what happens at the ends of the various movements between the positions shown in Figures 7.4 and 7.5. For example, \texttt{mv43()} is the code for the end of a movement of a pallet from position 4 to position 3. \texttt{assbreak()} and \texttt{mbreak()} are functions for when the assembly station or the machining station break down. \texttt{endmc()} and \texttt{endass()} are the event functions for when a pallet is done at either the machining or assembly station respectively.
7.4.4 The Results

According to the results, the CIM Lab can make approximately 25 robots and 25 cnc's in a 480 minute period. The assembly station has a 29 percent utilization, while it is reserved 66 percent of the time. The machining station has a 65 percent utilization, while it is reserved 80 percent of the time. With some modification of the scheduling and routing rules, these utilizations might be increased.

The CBST simulation model of the CIM Lab is very efficient and runs very quickly. If modular programming habits are used in the development of the program, it can be embellished easily.
8. Conclusions and Recommendations

The C-Based Simulation Toolkit (CBST) is a group of variables and data structures with sets of functions that operate upon them within a particular framework which supports discrete, continuous, and combined simulation. A simulation program using CBST consists of a C program with calls to the various CBST simulation support functions within the specified framework.

CBST provides support in the following areas:
(1) executive control, (2) system variable initialization, (3) list and entity management, (4) resource management, (5) random variate generation, (6) data collection, statistical computation, and reporting, (7) error handling, and (8) state variable management.

CBST is similar to other simulation packages of the approach 3 type in that it is basically a group of functions which support simulation and are called from a user written program in a general purpose programming language such as C, FORTRAN, or Pascal.

CBST is especially similar to GASP IV, DISC, and CSIM. CBST bears similarity to DISC and CSIM in that it based in the C programming language and supports discrete simulation. However, neither DISC nor CSIM provide support for continuous or combined simulation, whereas CBST does. CBST
employs the event scheduling strategy for next event selection, as does DISC, unlike CSIM, which uses the process interaction strategy.

CBST is most akin to GASP IV. Both support discrete, continuous, and combined simulation. Both utilize the event scheduling strategy for next event selection. The framework of functions and the manner in which the user must write simulation programs using CBST is fairly analogous to that in GASP IV.

CBST has some differences with GASP IV as well. Foremost of these differences is the base general purpose programming language. GASP IV is based in FORTRAN. This implies basic differences in the way functions are called, what they return, and the general way a program is written in CBST versus GASP IV. In addition, C is better suited for simulation than FORTRAN because it has dynamic memory management and custom made data types through structures.

Another major distinction of CBST from GASP IV is that CBST has a particular data structure and group of functions to deal with the management of resources, whereas GASP IV does not. These are the array of structures of type resnote and the functions: resavail(), rescapture(), resrelease(), resincr(), and resdecr(), which act upon the resnote structures. The functions test the availability of a
resource, capture or release units of a resource, and increase or decrease the number of available units of a resource. Information dealing with utilization, busy and idle times, and entity counts of the resources is automatically processed when using the resource management functions.

CBST also differs from GASP IV in the use of the differential equation solver. GASP IV uses the Runge-Kutta-England method, while CBST employs the Runge-Kutta-Fehlberg method. It should be noted that SLAM II (which has a version of GASP IV as a subset) also uses the Runge-Kutta-Fehlberg method.

Finally, the executive control functions of CBST are different. GASP IV has one function which handles discrete, continuous, and combined simulation control, while CBST employs three different functions. The method of advancing time in continuous and combined simulation is slightly different in CBST as well.

The enhancement of resource management functions and the fact that CBST is C-based provide it with more flexibility than GASP IV.

CBST may be further developed in several ways. The event list is currently kept on a doubly linked list. A
binary search tree data structure would lead to faster insertion onto the event list in cases where there are many event notices. The memory allocation functions could be modified to perform "garbage collection" strategies to increase the speed of a program. Functions which aid in simulation output reporting and analysis could be added. A version of CBST implementing the process view could be developed.

The primary uses of CBST are to model large and complex systems and to be a simulation driver for other software. In addition, CBST can be used to teach simulation, to examine different procedures in handling an event list, and to test different methods of analyzing simulation output.

CBST is well suited for large and complex simulation models. A large model might be a system with an immense number of entities, many different resources, and/or many different types of events. Since there is not very much overhead in memory space or in execution time when using CBST, large models can be accommodated. Complex models having elaborate decision rules such as routing rules or scheduling rules can be made using CBST. This is because the simulation program itself is a C program and thus the flexibility of the C language can be used to help model the complexity of the system.
CBST is designed to be a simulation driver for further software as well. Since it based in the language, C, it can be used in software programs written in C. Examples of possible software are graphical simulators, scheduling simulators, and other special purpose simulators. CBST was developed modularly so that it could be extended.

CBST would make a good teaching aid for a course in basic simulation modeling. The student would gain an understanding of the event scheduling view for discrete simulation. In addition, continuous and combined simulation concepts could be learned through the use of CBST.

Because CBST was developed in a modular fashion, the event list functions can be altered easily to implement other methods of handling the event list. CBST currently uses a doubly linked list for the event list. Alternatives for this include a circular list, and indexed list, and a binary search tree. CBST can be used to compare the execution of the various methods under various conditions.

CBST can be used to generate data to test various methods of analyzing simulation output data. Such methods include batch means and independent replications. Since a simulation program using CBST is a C program, these methods can be carried out within the program.
9. References


Appendices

A. Listing of CBST functions and variable declarations ........................................ 105

B. Listings of model programs ................................................................. 164
   1a. Single run M/M/1 queueing system model ....... 165
   1b. main() function for multiple runs ............... 170
   2. Cedar Bog Lake model ......................................................... 171
   3. Tank/truck model ............................................................. 175
   4. CIM Lab model ................................................................. 183

C. Outputs of models ................................................................. 210
   1a. Output of single run for M/M/1 model ............. 211
   1b. Output of multiple runs for M/M/1 model ...... 212
   2a. CBST output for Cedar Bog Lake model .......... 214
   2b. SLAM II output for Cedar Bog Lake model ...... 216
   3. Output for tank/truck model ......................... 219
Appendix A

Listing of CBST functions

/*
 * The C-Based Simulation Toolkit
 *
 * developed by
 *
 * Fazal U. Khan
 *
 */

/*
 * Influences on CBST
 * ------------------
 * 1. The GASP IV Simulation Language
 *    by A. Alan B. Pritsker
 *    - The overall framework of how simulation programs are
 *      written using CBST is fairly analogous to that in GASP IV
 *    - The CBST function cross() is very similar to the
 *      GASP IV function KROSS()
 *
 * 2. Data Structures and C Programs
 *    by Christoper J. Van Wyk
 *    - CBST functions insert(), makeelement(), and dumplist()
 *      are adapted from Van Wyk's C functions insert(),
 *      makeelement(), and dumpset(). Most of the variable names
 *      are the same.
 *
 * 3. Numerical Recipes in C: The Art of Scientific Programming
 *    by William H. Press, Brian P. Flannery,
 *    Saul A. Teukolsky, and William T. Vetterling
 *    - CBST functions ran3() and gaussian() are slightly modified
 *      versions of the C functions ran3() and gasdev() from
 *      Numerical Recipes in C. The CBST functions use the
 *      same names for the variables.
 *
 * 4. Numerical Analysis
 *    by Johnson and Riess
 *    - The CBST functions for the differential equation solver
 *      ( rkf(), gscal(), and desolve() ) are adapted from
 *      FORTRAN subroutines in Johnson and Riess. Most of the
 *      variable names are the same.
 */
/*
 * Breakdown of CBST functions:
 * 1. executive control
 * 2. system variable initialization
 * 3. list and entity management
 * 4. resource management
 * 5. random variate generation
 * 6. data collection, statistical computations, and reporting
 * 7. error handling
 * 8. state variable management
 *
 * 1. executive control
 * ------------------
 * discrete() - called by main() to implement discrete event simulation
 *
 * continuous() - called by main() for continuous algorithm
 *
 * combined() - called by main() to implement combined algorithm
 *
 * 2. system variable initialization
 * ------------------------------
 * discpinit() - initializes CBST variables to default values for discrete
 * simulation
 *
 * contpinit() - initializes CBST variables to default values for continuous
 * simulation
 *
 * combpinit() - initializes CBST variables to default values for combined
 * simulation
 *
 * initlist() - input: pointer to listnote
 * purpose: initialize list statistics
 *
 * 3. list and entity management
 * a. event list management
 * b. set (list) management
 * c. entity management
 * -----------------------------
 * 3a. event list management
 * ---------------------------
 * schedule() - takes as input type of event, time of event, and a
 * pointer to an entity and places this on the event list.
 * Also processes some statistics for the event list.
 */
nextevent() - purpose: removes the next event from the event list
return: pointer to the removed event

makeelement() - input: information and pointer to an entity
purpose: allocate memory for an element and place the
information and the entity pointer in the
element
return: pointer to the element

insert() - input: pointer to a list, pointer to an entity, and some
information
purpose: places the entity onto the list and rearranges
pointers in the list accordingly
return: pointer to a list

cancel() - input: integer representing event type to be removed
purpose: remove 1st occurrence of event type from the event list
return: 1, if found; else 0

cancelall() - input: integer representing event type to be removed
purpose: remove all occurrences of event type from the event
list
return: 1, if found; else 0

fixevlist() - used by cancel() and cancelall() to rearrange pointers
in the event list

3b. set (list) management

filefifo() - input: list number and pointer to entity
purpose: places an entity at the end of the file
indicated by the list number

filelifo() - input: list number and pointer to entity
purpose: places an entity at the beginning of the file
indicated by the list number

filelvf() - input: list number, pointer to entity, and ranking
attributes' indexes
purpose: to place an entity into a list according to
lowest value first order of the attribute(s)

filehvf() - input: list number, pointer to entity, and ranking
attributes' indexes
purpose: to place an entity into a list according to
highest value first order of the attribute(s)

remove() - low level function that takes as input the number of
the list from which the next entity is to be removed
and returns a pointer to the removed entity

removenum() - inputs: list number and rank of entity to be removed
        purpose: removes entity of specific rank from a list
        return: pointer to removed entity or NULL if not found

removespec() - inputs: list number and value being searched for
        purpose: removes entity with value in r[WR]
        return: pointer to removed entity or NULL if not found

fixlist() - inputs: list number and pointer to element
        purpose: used to rearrange pointers in a list

insfifo() - input: list number, pointer to entity, and time of entry
        purpose: creates an element to put entity at the end
                of the file indicated by the list number
        note: used by filefifo()

inslifo() - input: list number, pointer to entity, and time of entry
        purpose: creates an element to put entity at the beginning
                of the file indicated by the list number
        note: used by filelifo()

insprior() - similar to insfifo() and inslifo() except that
        it is used by filevfi() and filehvf() to rank entities in
        priority order

3c. entity management

makeentity() - allocates memory for an entity and returns a pointer to
        indicate the location

4. resource management

resavail() - input: resource number
        purpose: indicates number of units of the resource
                which are still available. Generates error
                message if capacity exceeded.
        return: units available

rescapture() - input: resource number
        purpose: updates stats. concerning a resource and
                captures one unit of a resource

resrelease() - input: resource number
        purpose: updates stats. concerning a resource and
                releases one unit of a resource
* resincr() - Inputs: resource number, number of units to increase capacity

* resdecr() - Inputs: resource number, number of units to decrease capacity

* 5. random variate generation

  * ran3() - uniform[0,1] (portable random number generator)
  * uniform
  * erlang
  * Weibull
  * normal
  * exponential
  * Erlang
  * gamma
  * beta
  * lognormal
  * Bernoulli
  * binomial
  * Poisson

* 6. data collection, statistical computations, and reporting

  * notevalue() - used to keep track of variables based on observation such as delay in queue

  * accumarea() - used to keep track of time-persistent variables such as number in queue

  * rresource() - to report statistics about a specific resource

  * rlist() - to report stats. about a specific list

  * replist() - to report stats. about all lists including the event list

  * clrres() - to clear stats. for a specific resource

  * clrlist() - to clear stats. for a specific list

  * clirov() - to clear stats. for an obav

  * clrtpl() - to clear stats. for a tpvar

  * bidarr() - Inputs: a pointer to a vector and a time value purpose: to build an array of values to be plotted later

  * getarray() - allocates space in the heap for the array of state variables that are recorded by bidarr()
cplot() - purpose: to plot variables against an independent variable
(usually time) based on contents of the array built by bidarr()

pflestat() - used to update statistics concerning a particular file
when entity is placed onto the file

mflestat() - inputs: list number (int); entity's entry time (double)
purpose: used to update stats concerning a file when
an entity is taken off of the file

7. error handling
-----------
perror() - prints an error message and aborts program execution

perror1() - inputs: 1 char. string and an integer
purpose: prints an error message combining the char. string
and the integer, and aborts program execution

perror2() - inputs: 2 char. strings and an integer
purpose: prints an error message combining the char. strings
and the integer, and aborts program execution

checkt() - checks values of tdrec, hmax, hmin, and tolr

8. state variable management
-------------------------------
desolve() - inputs: various parameters needed to solve the
differential equation(s)
purpose: to solve ODE's by using rkf() to present a
solution and using gamcal to check if the
accuracy is acceptable and adjusting parameters
if need be

gamcal() - purpose: calculates and returns a value, gamma,
based on input parameters. Gamma is used to
determine whether accuracy is met for ODE's.
Called by desolve().

rkf() - purpose: performs variable step Runge-Kutta-Fehlberg fourth and
fifth order methods for solving a system of ODE's.
The fifth order method is used to estimate the error.
Called by desolve().
cross() - inputs: crossing variable, threshold variable, tolerance, and direction of crossing
purpose: to monitor variables, watching for crossing of thresholds. (Used in stevcond() )

advance() - inputs: start time, and time
purpose: to advance from one time to another. Used by continuous() and combined().
- calls RKF algorithm
- finds state event if needed
- sends control to sevents() if needed

update() - updates state variables and their rate of change

User written functions that are needed when using CBST

main() - used to start simulations by calling one of the executive control routines

uinit() - user initializes variables to other than default values and/or initializes non-CBST variables

stopcheck() - to check for stopping conditions

- discrete
---
sevents() - used to call proper time-event routines

(time-event codes) - purpose: to implement logic associated with each time-event

- continuous
---
yfirst() - contains the differential equation formulas in a certain format

stevcond() - purpose: to set conditions for a state event and to set the vector that shows which state event(s) has occurred.

sevents() - checks a vector to see if state-event routines should be called

(time-event codes) - purpose: to take specific actions when a specific state event is encountered
typedef struct element element;
struct element {
    element *after;
    element *before;
    int evtype;
    double evtime;
    double entry;
    entity *ent;
};

struct tvar {
    double ttot; /* current time */
    double tlast; /* time of last update */
    double tclr; /* time stats cleared */
    double mean; /* mean of var */
    double areah; /* accumulated area */
}
double harea; /* accum. area of h^2 */
);

typedef struct tpvar tpvar;

struct obsvar {
  double num;  /* number of obs. */
  double mean; /* mean of var */
  double sum; /* sum of observations */
  double sumx; /* sum of obs^2 */
};

typedef struct obsvar obsvar;

struct listnote {
  char *name;
  int lmax;
  int lcur;
  tpvar length;
  obsvar wait;
};

typedef struct listnote listnote;
listnote evb; /* to keep stats for event list */
listnote lb[NLISTS]; /* to keep stats for queues */
element *list[NLISTS];
element *tTail[NLISTS];
element *evlist = NULL;

struct resnote {
  char *name; /* name tag */
  int cap; /* capacity */
  int utilcur; /* current util */
  tpvar util; /* utility stats. */
  double busy; /* busy time counter */
  boolean bflag; /* busy flag */
  double idle; /* idle time counter */
  double idtimemax; /* max idle time */
  double bsytimax; /* max busy time */
  long int entcount; /* entity count */
};

typedef struct resnote resnote;
resnote rn[NRES];

double tnow; /* updated at event times */
double tdfull; /* max step size */
double tdrec; /* step size for recording */
double hmin; /* min h in desolve() */
double hmax; /* max h in desolve() */
double tolr; /* tolerance used in gamcal() */

int sfleg[NSTEVF];

double dnow[NODE];
double dlast[NODE];
double snow[NODE];
double slast[NODE];

int seed[NSEEKS];
int riff[NSEEKS];

/* function declarations */

/* executive control */
void discrete(), continuous(), combined();

/* initialization */
void compinit(), continit(),
discinit(), initlist();

/* list and entity management */
entity *makeentity();
void dumplist();
element *insert();
void insprior(), filelvf(), filehvf();
element *makeelement();
element *nextevent();
int cancel(), cancelall();
void fixevlist(), fixlist();
void schedule();
entity *removelist(), *removenum(), *removespec();
void infifo(), inmifo(), filefifo(), filelifo();

/* resource management */
void resincr(), resdecr(), rescapture(), resrelease();
int resavail();

/* random deviate generation */
double ran3(), normal(), erlang(), weibull();
double gaussian(), gam1(), gam2(), gamma();
double expon(), beta(), betal(), uniform();
double triang(), triang1(), lognormal();
int bernoulli(), binomial(), poisson();

/* data collection, statistical */
/* computation, and reporting */
void pfisetstat(), mfisetstat();
void notevalue(), accumarea();
void clriset(), clrres(), clrtip(), clrov();
void rlist(), rresource(), replist();
void rhead();
void bidarr(), cplot();
double **getarray();
void gettptstat(), gettovstat();

/* error handling */
void perror(), perror1(), perror2(), checkt();

/* state variable management */
double *rknf(), gmanal();
boolean desolve();
void advance(), update();
int cross();

void diagres(), diaglist();
double whennext();

/*
 * executive control functions
 */
void discrete()
{
    int check;
    element *pel;

    /* initialize CBST variables */
discpinit();

    /* initialize user variables */
ulinit();

    for(;;)
    {
        /* check for stopping conditions */
        if (check = stopcheck())
            return;
    }
/* make sure event list is not empty */
if (empty(evlist))
        perror("discrete(): event list is empty");

/* take next event from event list */
pel = nextevent();
if (!pel)
        perror("discrete(): error in getting next event from event file");

/* set the current simulation clock time */
/* to time of event from removed event */
/* notice */
tnow = pel->evtime;

/* call user written tevents() and send */
/* the pointer to removed event notice */
tevents(pel);

/* free the memory of the removed event */
/* notice */
free(pel);
}

void continuous()
{
    double tt2, tt3, tbes, tend, dx23;
    int ff2, ff3, check;
    double tenst; /* time of end of next step */
    double tinenst; /* time of end of last step */
    double tntrec; /* next recording time */
    double tinrec; /* last recording time */
    double tres; /* resolution */

    printf("Continuous Simulation
");   
    printf("with the C-Based Simulation Toolkit
");       
    printf("developed by Fazal U. Khan (1990)
");   

    /* initialize variables */
    continit();
    uninit();

    /* initialize flags and times */
    ff2 = ff3 = 1;
    tenst = tntrec = tnow;
tenst = tinrec = 0.0;
tres = tdfull/100.;
checkt();

/* record initial values of state variables */
bidarr(snow, tnow);

while(1)
{
  tbeg = tnow;

  /* check for stopping conditions */
  if (check = stopcheck())
    return;

  /* establish tend and set flag values */
  if (ff2)
    {
    tlenst = tenst;
    tenst = tlenst + tdfull;
    }

  if (ff3)
    {
    tinrec = tinrec;
    tinrec = tinrec + tdrec;
    }

  tt2 = tenst;
  tt3 = tinrec;

  dd23 = ABS(tt2 - tt3);

  if (dd23 < tres )
    {
    ff2 = 1;
    ff3 = 1;
    tend = tt3;
    }
  else /* dd23 > tres */
    {
    if (tt3 < tt2)
      {
      tend = tt3;
      ff2 = 0;
      ff3 = 1;
      }
    else /* tt2 < tt3 */
      {
      tend = tt2;
      ff2 = 1;
ff3 = 0;
}

/* tend is set. Try to advance simulation clock */
/* time from tbeg to tend. advance() will move */
/* the simulation clock time forward while */
/* updating the state variable values. It will */
/* check for state-events and find the time of */
/* the first state-event if necessary. It will */
/* call user written aevents() to execute the */
/* proper state-event code */

advance(tbeg, tend);

if ( (tend - tnow) > tres )
{
    /* simulation clock time was not advanced to */
    /* tend. Set flags accordingly */
    ff2 = 0;
    ff3 = 0;
}
else if (ff3)
{
    /* simulation clock time was advanced to tend */
    /* and recording of state variable values is */
    /* to occur at this time */
    bidarr(snow, tnow);
}

}

void combined()
{
    double tt1, tt2, tt3, tbeg, tend, add2, dom3;
    int ff1, ff2, ff3;
    element *p;
    double tenst;   /* time of end of next step */
    double tenst;   /* time of end of last step */
    double trec;    /* next recording time */
    double tlnrec;  /* last recording time */
    double tres;    /* resolution */

    printf(" Combined Simulation\n\n");

118
printf(" with the C-Based Simulation Toolkit\n\n");
printf(" developed by Fazal U. Khan (1990)\n\n");

/* initialize variables */
comptinit();
uninit();

/* initialize flags and times */
ff1 = ff2 = ff3 = 1;
tenst = tinrec = tnow;
tlenst = tinrec = 0.0;
tres = tdfull/100.;
checkt();

/* record initial values of state variables */
bladarr(snow, tnow);

while(1)
{
  tbeg = tnow;

  /* check for stopping conditions */
  if (stopcheck())
    return;

  /* establish tend and set flag values */

  /* check time of next event vs. end of */
  /* next step */
  if (event != NULL)
    tt1 = thenext();
  else
    tt1 = tnow + 5.0*tdrec;

  if (ff2)
  {
    tlenst = tenst;
    tenst = tlenst + tdfull;
  }

  if (ff3)
  {
    tinrec = tinrec;
    tinrec = tinrec + tdrec;
  }

  tt2 = tenst;
  tt3 = tinrec;
  dd12 = ABS(tt1 - tt2);

  /*...*/
if ( dd12 < tres )
  {
    tend = tt1;
    ff1 = 1;
    ff2 = 1;
    ff3 = 0;
  }
else
  {
    if (tt1 < tt2)
      {
        tend = tt1;
        ff1 = 1;
        ff2 = 0;
        ff3 = 0;
      }
    else /* tt1 > tt2 */
      {
        tend = tt2;
        ff1 = 0;
        ff2 = 1;
        ff3 = 0;
      }
  }

ddt3 = ABS(tend - tt3);

if (ddt3 < tres )
  ff3 = 1;
else /* ddt3 > tres */
  {
    if (tt3 < tend)
      {
        tend = tt3;
        ff1 = 0;
        ff2 = 0;
        ff3 = 1;
      }
  }

/* tend is set. Try to advance simulation clock */
/* time from tbeg to tend. advance() will move */
/* the simulation clock time forward while */
/* updating the state variable values. It will */
/* check for state-events and find the time of */
/* the first state-event if necessary. It will */
/* call user written sevents() to execute the */
/* proper state-event code */

advance(tbeg, tend);

120
if ( (tend - tnow) > tres )
{
    /* simulation clock time was not advanced to */
    /* tend. Set flags accordingly */
    ff1 = 0;
    ff2 = 0;
    ff3 = 0;
}
else
{
    /* simulation clock time was advanced to tend */
    if (ff3)
    {
        /* record state variable values if flag */
        /* so indicates */
        bldarr(snow, tnow);
    }
    if (ff1)
    {
        /* carry out time event logic if flag is */
        p = nextevent();
        if (p)
            perror("combined(): event list was empty");
        tevents(p);
        free(p);
    }
}
/*
 * system variable initialization functions
 */

void discplnit()
{
    int k;

    /* initialize all time variables */
    tnow = 0.;

    }
/* init some resource stats */
for (i=0; i<ARES; i++)
{
    rn[i].name = "";
    rn[i].cap = 0;
    rn[i].util = 0;
    rn[i].busy = 0.;
    rn[i].bflag = 0;
    rn[i].idle = 0.;
    rn[i].idel = 0.;
    rn[i].beytim = 0.0;
    rn[i].beytimax = 0.0;
    rn[i].entcount = 0;
    clrtp ( &rn[i].util );
}

/* init some list statistics */
for (i=0; i<NLISTS; i++)
{
    lb[i].name = NULL;
    initlist(&lb[i]);
    list[i] = NULL;
    tail[i] = NULL;
}

/* init event list statistics */
evlist = NULL;
evbev = NULL;
evbev.name = "event list";
initlist(&evbev);

/* init seeds, etc. for ran3() */
for (i=0; i<NSNDSEEDS; i++)
rif[i] = 0;


void compinit()
{
    int i;

    /* initialize all time variables */
tnow = 0.0;
tdfull = 0.0;
tdrec = 0.0;

    /* initialize rkf variables */
hmin = hmax = tolr = 0.0;

    /* init state event flags */
    /* note: for combined sim. only */
for (i=0; i<NSTEVF; i++)
    sflag[i] = 0;

/* init seeds, etc. for ran3() */
for (i=0; i<NSEEDS; i++)
    riff[i] = 0;
}

void combplinit()
{
    int i;

    /* initialize all time variables */
    tnow = 0.;
    tdfull = 0.;
    tddrec = 0.;

    /* initialize rkf variables */
    hmin = hmax = toir = 0.0;

    /* init some resource stats */
    for (i=0; i<NRES; i++)
    {
        rn[i].name = " ";
        rn[i].cap = 0;
        rn[i].utilcur = 0;
        rn[i].bus = 0.;
        rn[i].bf = 0;
        rn[i].idle = 0.;
        rn[i].idtime = 0.;
        rn[i].bf = 0;
        rn[i].idttime = 0.;
        rn[i].entcount = 0;
        cirtp( &rn[i].util); }

    /* init state event flags */
    /* note: for combined sim. only */
    for (i=0; i<NSTEVF; i++)
        sflag[i] = 0;

    /* init some list statistics */
    for (i=0; i<NLISTS; i++)
    {
        lb[i].name = NULL;
        initlist(&lb[i]);
        list[i] = NULL;
        tail[i] = NULL;
    }
}
/* init event list statistics */
evlist = NULL;
evlist = "event list";
initlist(&evb);

/ * init seeds, etc. for ran3() */
for (i=0; i<10; i++)
    riff[i] = 0;
}

void initlist(k)
listnote *k;
{
    k->lmax = 0;
    k->lcur = 0;
    clrtp( &k->length );
    clrov( &k->wait );
}

/ *
   * list and entity management functions
   */

void schedule(s, b, pen)
int s;
double b;
entity *pen;
{
    double dt, faz1;

    evlist = insert(evlist, a, b, tnow, pen);

    faz1 = (double) evb.lcur;
    accumarea(faz1, &evb.length, tnow);

    evb.lcur += 1;
    if ( evb.lcur > evb.lmax )
        evb.lmax = evb.lcur;
}
element *nextevent()
{
    element *p;
    double dt, faz1;

    /* make sure event list has something on it */
    if (empty(evlist))
        return NULL;

    /* assign pointer to list element on event */
    /* list to p and reassign the following */
    /* element as new first */
    p = evlist;
    evlist = p->after;
    if (evlist == NULL)
        evlist->before = NULL;

    /* revise stats. about length */
    faz1 = (double) evb.lcur;
    accumarea(faz1, &evb.length, p->evtime);
    evb.lcur -= 1;

    /* revise stats. about waiting times */
    dt = p->evtime - p->entry;
    notevalue(dt, &evb.wait);

    /* return pointer to removed event record */
    return p;
}

int cancel1(type)
int type;
{
    element *p;
    double dt, faz1;

    if (empty(evlist))
        return 0;

    p = evlist;
    while (p->after && p->evtype != type)
        p = p->after;

    if (p->evtype == type)
        /* event of this type is on the event list */
        { //
            fixevlist(p);

        return 1;
    }

    return 0;
}
/* revise stats. about length */
fa1 = (double) evb.lcur;
accumulate(fa1, &(evb.length), &now);
evb.lcur -= 1;

/* revise stats. about waiting times */
dt = now - p->tentry;
notetime(dt, &(evb.wait));

free(p);
return 1;
)

else if (p->after == NULL)
     /* event of this type is not on event list */
     return 0;

)

void fixevlist(p)
element *p;
{
    if (p->before == NULL)
        /* notice is at head of event list */
        {
            if (p->after)
                {
                    evlist = p->after;
evlist->before = NULL;
                }
            else
                evlist = NULL;
        }
    else if (p->before && p->after)
        {
            /* notice is in middle of event list */
            p->before->after = p->after;
p->after->before = p->before;
        }
    else if (p->after == NULL)
        {
            /* notice is at end of the event list */
p->before->after = NULL;
        }
}
int cancelall(type)
int type;
{
    element *p, *doomed;
    double dt, fzl1;

    if (empty(evlist))
        return 0;

    doomed = NULL;
p = evlist;
while (p)
{
    if (p->evtype == type)
        /* event of this type is on the event list */
        
        fixevlist(p);

        /* revise stats. about length */
fzl1 = (double) evb.lcur;
    accumulate(fzl1, &(evb.length), tnow);
evb.lcur -= 1;

    /* revise stats. about waiting times */
dt = tnow - p->tentry;
    notevalue(dt, &(evb.wait));

    doomed = p;
    free(doomed);
}
p = p->after;
}

if (doomed)
    return 1;
else
    /* event of this type is not on event list */
    return 0;
}

element *makeelement()
{
    element *p;

    return NULL;
}
p = (element *) calloc(1, sizeof(element));
if (!p)
    perror("makeelement(): calloc() failed");
return p;
}

element *insert(s, n, x, y, ptrent)
    element *s;
    int n;
    double x;
    double y;
    entity *ptrent;
{
    element dummy, *p, *new;
    p = &dummy;
    p->after = s;
    while(p->after && p->after->evtime <= x)
        p = p->after;
    if (!p)
        perror("insert(): impossible p");
    if (! (p->after && p->after->evtime == x))
    {
        /* create the event notice and assign its values */
        new = makeelement();
        new->evtype = n;
        new->evtime = x;
        new->entry = y;
        new->entry = ptrent;

        /* place the event notice in the event list */
        new->after = p->after;
        if (new->after)
            new->after->before = new;
        p->after = new;
        if (new == dummy.after)
            new->before = NULL;
        else
            new->before = p;
    }
    return dummy.after;
}

void filefifo(n, c)
int n; /* list number */
entity *c; /* pointer to entity entering list */
{
    if (n > WLISTS)
        perror("filefifo(): the list number equals or exceeds WLISTS, ", n);

    /* insert the new element at the end of the queue */
    insfifo(n, tnow, c);

    /* update file statistics */
    pfilestat(n);
}

void filefifo(n, c)
int n; /* list number */
entity *c; /* pointer to entity entering list */
{
    if (n > WLISTS)
        perror("filefifo(): the list number equals or exceeds WLISTS, ", n);

    /* insert the new element at the end of the queue */
    insfifo(n, tnow, c);

    /* update file statistics */
    pfilestat(n);
}

void filelvf(n, c, n2)
int n; /* list number */
entity *c; /* pointer to entity entering list */
int n2; /* attribute number used to rank entities */
{
    if (n > WLISTS)
        perror("filelvf(): the list number equals or exceeds WLISTS, ", n);

    /* insert the element lowest value first based on */
    /* attributes r[0]...r[n2] */
    insprior(n, tnow, c, n2, "lvf");

    /* update file statistics */
    pf filestat(n);
}
void filehvi(n, c, n2)
    int n;    /* list number */
    entity *c; /* pointer to entity entering list */
    int n2;    /* attribute number used to rank entities */
{
    if (n > NLISTS)
        perror("filehvi(): the list number equals or exceeds NLISTS, ", n);

    /* insert the element highest value first based on */
    /* attributes r[0]...r[n2] */
    insprior(n, now, c, n2, "hvi");

    /* update file statistics */
    pfillestat(n);
}

void insprior(n, x, q, n2, dir)
    int n;    /* list number */
    double x; /* time of entry into queue */
    entity *q; /* pointer to entity entering queue */
    int n2;    /* index of last attribute to base ranking */
    char *dir; /* string indicating lvf or hvf priority */
{
    element *new, *p, dummy, *beg, *end;
    int ii, iflag;

    /* create element and assign pointer to 'new' */
    new = makeelement();
    new->evtype = n;
    new->tentry = x;
    new->pent = q;

    /* if list is empty, assign head and tail pointers */
    if (empty(list[n]) )
    {
        list[n] = new;
        tail[n] = new;
    }
    else
    {
        /* list is not empty so the ranking procedure */
        /* must be used */
        beg = &dummy;
        beg->before = NULL;
    }
beg->after = list[n];
end = ltail[n];

iflag = 1;
ii = 0;

while ( ii < n2 && iflag )
  {
    iflag = 0;
    if (dir[0] == 'l')
      while (beg->after && (beg->after->pent->r[ii] < q->r[ii])
             && (beg != end))
        {
          beg = beg->after;
          iflag = 1;
        }
    else if (dir[0] == 'r')
      while (beg->after && (beg->after->pent->r[ii] > q->r[ii])
             && (beg != end))
        {
          beg = beg->after;
          iflag = 1;
        }
    else
      perror("insprior(): unexpected string for dir");

    p = beg;

    while (p->after && (p->after->pent->r[ii] == q->r[ii])
           && (p != end))
      {
        p = p->after;
        iflag = 1;
      }

    end = p;
    ii++;
  }

/* insert the element */
new->after = p->after;
if (new->after)
  new->after->before = new;
p->after = new;
if (new == dummy.after)
  new->before = NULL;
else
  new->before = p;

/* reassign head and tail pointers */
void fixlist(n, p)
    int n;  /* list number */
    element *p;
    
    if (n > NLISTS)
        perror("fixlist(): the list number equals or exceeds NLISTS, ", n);
    
    if (p->before == NULL)
        /* notice is at head of list */
        
        if (p->after)
            
            list[n] = p->after;
            list[n]->before = NULL;
        
        else
            
            list[n] = NULL;
            tail[n] = NULL;
        
    else if (p->before && p->after)
        
        /* notice is in middle of list */
        p->before->after = p->after;
        p->after->before = p->before;
    
    else if (p->after == NULL)
        
        /* notice is at end of the list */
        tail[n] = p->before;
        p->before->after = NULL;
    
);
double x; /* time of entry into queue */
entity *p; /* pointer to entity entering queue */
{
    element *new;

    /* create element and assign pointer to 'new' */
    new = makeelement();
    new->evtype = n;
    new->evtime = new->tentry = x;
    new->pent = p;

    /* if list is empty, assign pointer to new */
    /* element to the head and tail pointers */
    if ( empty(list[n]) )
    {
        list[n] = new;
        ltail[n] = new;
    }
    else
    {
        /* place new element at the end of the queue */
        /* and update pointers */
        ltail[n]->after = new;
        new->before = ltail[n];
        ltail[n] = new;
    }
}

void inslifo(n, x, p)
int n; /* list index */
double x; /* time of entry into queue */
entity *p; /* pointer to entity entering queue */
{
    element *new;

    /* create element and assign pointer to 'new' */
    new = makeelement();
    new->evtype = n;
    new->evtime = new->tentry = x;
    new->pent = p;

    /* if list is empty, assign pointer to new */
    /* element to the head and tail pointers */
    if ( empty(list[n]) )
    {
        list[n] = new;
        ltail[n] = new;
    }
else
{
    /* place new element at the end of the queue */
    /* and update pointers */
    list[n]->before = new;
    new->after = list[n];
    list[n] = new;
}

entity *removelst(n)
int n;        /* input list number */
{
    element *d;
    entity *khan;
    double dt, faz1;

    if (n > NLISTS)
        perror("removelst(): the list number equals or exceeds NLISTS, ", n);

    if (empty(list[n]) )
        return NULL;

    d = list[n];
    fixlist(n, d);

    while (d->ent->tentry);

    khan = d->ent;
    free(d);

    return (khan);        /* return ptr to entity */
}

entity *removenum(n, rank)
int n;        /* input list number */
int rank;     /* rank on list*/
{
    element *d;
    double dt, faz1;
    entity *khan;
    int f;

    if (n > NLISTS)
        perror("removenum(): the list number equals or exceeds NLISTS, ", n);

    d = list[n];
    fixlist(n, d);

    /* find entry */
    while (d->ent->tentry);

    khan = d->ent;
    free(d);

    return (khan);        /* return ptr to entity */
}
if (empty(list[n]) )
    return NULL;

if (rank < 1)
    perror("removenum(): cannot specify a rank less than 1");

if (rank == 1)
{
    khan = remove1st(n);
    return khan;
}

d = list[n];

for (i=1; i<rank; i++)
{
    d = d->after;
    if (d)
    
        return NULL;

    fixlist(n, d); /* rearrange pointers in list n */

    mfiestat(n, d->tentry);

    khan = d->pent;
    free(d);

    return (khan); /* return ptr to entity */
}

entity *removespec(n, x)
int n;    /* input list number */
double x; /* spec of entity */
{
    element *d;
    double dt, fa1;
    entity *khan;

    if (n > NLISTS)
        perrorl("filefio(): the list number equals or exceeds NLISTS, ", n);

    if (empty(list[n]) )
        return NULL;

    d = list[n];
    while (d->after && d->pent->r[KR] != x)
        d = d->after;

    d = d->after;

    return d; /* return ptr to entity */
}
if (!d->after) {
    return NULL;
} else {
    fi(xlist(n, d)); /* rearrange pointers in list n */

    mfencode(n, d->entry);

    khan = d->pent;
    free(d);

    return (khan); /* return ptr to entry */
}

entity *makeentity()
{
    entity *p;

    p = (entity *) calloc(1, sizeof(entity));
    if (!p)
        perror("makeentity(): calloc() failed");
    return p;
}

void dumplist(s)

    element *s;

    {  
        while (s)
            {
                printf("%lu %10.3f bef%lu aft%lu \n", s, s->evtime, s->before, s->after);
                s = s->after;
            }
        printf("\n");
    }

/*
 * resource management functions
 */
int resavail(num)
int num;
{
   /* function to test availability of a resource */
   /* - returns number of available servers if there */
   /* are any */
   /* - returns 0 if resource is at capacity */
   /* - generates an error message if number of */
   /* servers being used is greater than capacity */
   if (num >= NRES)
      perror1("resavail(): resource number exceeds or equals NRES, ", num);
   if (rn[num].utilcur < rn[num].cap)
      return (rn[num].cap - rn[num].utilcur);
   else if (rn[num].utilcur == rn[num].cap)
      return 0;
   else
      perror2("resavail(): capacity of resource",num,"exceeded");
}

void rescapture(num, units)
int num;
int units;
{
   double faz1, dt;
   /* function to capture units of a resource and to */
   /* update the stats. associated with that resource */
   if (num >= NRES)
      perror1("rescapture(): resource number exceeds or equals NRES, ", num);
   /* check if resource is available. If so, start */
   /* updating stats. If not, generate an error message */
   if (resavail(num) >= units)
   {
      /* increment entity count */
      rn[num].entcount++;
      /* update time persistent statistic of */
      /* utilization */
      faz1 = (double) rn[num].utilcur;
      accumarea(faz1, &(rn[num].util), tnow);
   
   
}

137
/* update stats on idle time if server */
/* was idle */
dt = rn[num].util.ttot - rn[num].util.tlast;
if (rn[num].utilcur == 0 && dt > 0.0)
{
    rn[num].idle = dt;
    if (rn[num].idle > rn[num].idlimax )
        rn[num].idlimax = rn[num].idle;
}

/* update stats on busy time if server */
/* was busy */
if (rn[num].util.ttot == rn[num].util.tlast)
    rn[num].bflag = 1;
else
    rn[num].bflag = 0;

/* increase number being utilized by 'units' */
rn[num].utilcur += units;
else
    perror2("rescapture(): resource",num, "requested over capacity");

}

void resrelease(num, units)
int num;
int units;
{
    double dt, faz1;

    /* function to update stats on a resource when service */
    /* is completed */

    if (num >= NRES)
        perror1("resrelease(): resource number exceeds or equals NRES, ", num);

    if ( units <= rn[num].utilcur )
    {
        /* update time persistent statistic of */
        /* utilization */
        faz1 = (double) rn[num].utilcur;
        accumarea(faz1, &(rn[num].util), tnow);

        /* update stats on busy time */
        dt = rn[num].util.ttot - rn[num].util.tlast;
        if (rn[num].bflag == 1 && rn[num].utilcur == rn[num].cap )
            rn[num].busy += dt;
        else if (rn[num].utilcur == rn[num].cap )
            rn[num].busy = dt;
        else
            if (rn[num].busy > rn[num].beylimax)
rn[num].barytimmax = rn[num].busy;

/* decrease number being utilized by 1 */
rn[num].utilcur -= units;
}
else
    perror2("resrelease(): amount of resource", num,
        "released makes current units available exceed capacity");
}

void resdecr(num, units)
int num, units;
{
    int fazi;

    /* function to decrease the capacity of a resource */

    if (num >= NRES)
        perror1("resdecr(): resource number exceeds or equals NRES, ", num);

    fazi = rn[num].cap - rn[num].utilcur - units;

    if (fazi >= 0)
        rn[num].cap -= units;
    else
        perror2("resdecr(): resource", num,
            "decreased below zero capacity");
}

void resincr(num, units)
int num, units;
{
    /* function resincr() increases the capacity of a */
    /* resource by units */

    if (num >= NRES)
        perror1("resincr(): resource number exceeds or equals NRES, ", num);

    rn[num].cap += units;
}

/*
 * random variate generation functions
 */
```c
#define MBIG 10000000000
#define MSEED 161803398
#define NZ 0
#define FAC (1.0/MBIG)
#define PI 3.141592654
#define E 2.718281828

int bernoulli(n, p)
{
    int n; /* r.n. stream */
    double p; /* chance of success */
    
    double u;
    
    if ( (u = ran3(n)) <= p )
        return 1;
    else
        return 0;
}

int binomial(n, t, p)
{
    int n; /* r.n. stream */
    int t; /* number of trials */
    double p; /* chance of success */
    
    int i;
    int x = 0;
    
    for (i=1; i<=t; i++)
        x = x + bernoulli(n, p);
    
    return x;
}

int poisson(n, mean)
{
    int n; /* r.n. stream */
    int mean; /* mean (num expected) */
    
    double a, b;
    int i;
    double u, lp1;
    
    a = exp( - (double) mean );
    b = 1.0;
    i = 0;
    
    while(1)
    
    return i;
```
u_ip1 = ran3(n);
b *= u_ip1;
if ( b < a )
    return 1;
else
    i++;
}

double ran3(n)
int n;
{

static int inext[NSEEDS], inextp[NSEEDS];
static long m[n][NSEEDS][56];
long mj, mk;
int i, ii, k;
int *idum;

idum = &seed(n);
if (*idum < 0 || riff[n] == 0)
{
    riff[n] = 1;
mj = NSEED - (*idum < 0 ? -*idum : *idum);
mj %= MBIG;
m[n][55] = mj;
mk = 1;
for (i=1; i<=54; i++)
    {
    ii = (21 * i) % 55;
m[n][ii] = mk;
mk = mj - mk;
    if (mk < MZ)
        mk += MBIG;
mj = m[n][ii];
    }
for (k=1; k<=4; k++)
    for (i=1; i<=55; i++)
        {
        m[n][i] -= m[n][(i+(i+30) % 55)];
        if (m[n][i] < MZ)
            m[n][i] += MBIG;
        }
inext[n] = 0;
inextp[n] = 31;
*idum = 1;
If (++inext[n] == 56)
    inext[n] = 1;

if (++inextp[n] == 56)
    inextp[n] = 1;
    mj = ma[n][inext[n]] - ma[n][inextp[n]];
if (mj < M2)
    mj += M16;
    ma[n][inext[n]] = mj;
return mj*FAC;

double uniform(n, lo, hi)
int n;
double lo; /* low number in interval */
double hi; /* high number in interval */
{
    /* this function returns a uniform[lo, hi] random variate */

    return (lo + (hi - lo)*ran3(n));
}

double triang(n, lo, hi, mid)
int n;
double lo;
double hi;
double mid;
{
    double d;

    d = (mid - lo)/(hi - lo);
    return (lo + (hi - lo)*triang1(n, d));
}

double triang1(n, c)
int n;
double c; /* mode in a [0,1] triangular dist. */
{
    /* This function generates a triang(0, 1, c) random deviate */
    /* according to the procedure in Law and Kelton on page 261. */

    double u;
\[
\text{double beta(n, a, b, alpha1, alpha2)} \\
\text{int n;} \\
\text{double a;} \quad /* \text{beginning of interval */} \\
\text{double b;} \quad /* \text{end of interval */} \\
\text{double alpha1;} \quad /* \text{shape parameter 1 */} \\
\text{double alpha2;} \quad /* \text{shape parameter 2 */} \\
\{ \\
\quad /* \text{This function generates a } \text{beta(alpha1, alpha2) */} \\
\quad /* \text{random deviate on the interval [a,b] according */} \\
\quad /* \text{to the procedure on page 260 of Law and Kelton. */} \\
\quad double x; \\
\quad x = \text{beta1(n, alpha1, alpha2);} \\
\quad \text{return (a + (b-a)*x);} \\
\}
\]

\[
\text{double beta1(n, alpha1, alpha2)} \\
\text{int n;} \\
\text{double alpha1;} \quad /* \text{shape parameter 1 */} \\
\text{double alpha2;} \quad /* \text{shape parameter 2 */} \\
\{ \\
\quad /* \text{This function generates a } \text{beta(alpha1, alpha2) */} \\
\quad /* \text{random deviate on the interval [0,1] according */} \\
\quad /* \text{to the procedure on page 260 of Law and Kelton. */} \\
\quad double y1, y2; \\
\quad y1 = \text{gamma(n, alpha1, 1.0);} \\
\quad y2 = \text{gamma(n, alpha2, 1.0);} \\
\quad \text{return (y1/(y1+y2));} \\
\}
\]

\[
\text{double gamma(n, alpha, beta)} \\
\text{int n;} \\
\text{double alpha;} \quad /* \text{shape parameter */} \\
\]
```c
double beta; /* scale parameter */
{
    /* This function generates a gamma(alpha, beta) */
    /* random deviate for alpha > 0 and beta > 0. */
    /* This function calls either gam1, gam2, or expon */
    /* depending upon the value of alpha. */
    if ( alpha < 1.0 )
        return (beta * gam1(n, alpha));
    else if (alpha > 1.0 )
        return (beta * gam2(n, alpha));
    else /* alpha == 1.0 */
        return (beta * expon(n, 1.));
}

double gam1(n, alpha)
int n;
double alpha; /* shape parameter */
{
    /* This function generates a gamma(alpha, 1) random */
    /* deviate for the case where alpha < 1 as described */
    /* in Law and Kelton (pp. 255-258). Note that alpha */
    /* is assumed to be greater than 0 */
    double u1, u2, b, p, y;
    b = (E + alpha)/E;
    while(1)
    {
        u1 = ran3(n);
        p = b * u1;
        if ( p > 1.0 )
        {
            y = -log((1-p)/alpha);
            u2 = ran3(n);
            if (u2 <= pow(y, (alpha - 1.0)) )
                return y;
        }
        else /* p is le 1.0 */
        {
            y = pow(p, (1.0/alpha));
            u2 = ran3(n);
            if (u2 <= exp(-y))
                return y;
        }
    }
}
```

double gam2(n, alpha)
int n;
double alpha; /* shape parameter */
{
    /* This function generates a gamma(alpha, 1) random */
    /* deviate for the case where alpha > 1 as described */
    /* in Law and Kelton (pp. 255-258) */

double s, b, q, th, d, u1, u2, v, y, z, w;

    a = pow((2.0*alpha - 1.0), -0.5);
b = alpha - log(4.0);
q = alpha + 1.0/alpha;
th = 4.5;
d = 1.0 + log(th);

    while(1)
    {
        u1 = ran3(n);
u2 = ran3(n);

        v = a * log(u1/(1-u1));
y = alpha * exp(v);
z = u1*u1*u2;
w = b + q*v - y;

        if ( (w + d - th*z) >= 0.0 )
            return y;
        if ( w >= log(z) )
            return y;
    }
}

double expon(n, beta)
int n;
double beta; /* mean */
{
    return (-beta*log(ran3(n))) ;
}

double weibull(n, alpha, beta)
double alpha; /* shape parameter */
double beta; /* scale parameter */
{
    return pow((log(ran3(n))), (1./alpha)) * beta;
double erlang(n, beta, m)
int n;
double beta; /* mean */
int m; /* number of exponential samples */
{
    /* implementation of m-Erlang random deviate */
    /* generator as described in Law and Kelton */
    /* on page 254 */

double u2 = 1.0;
int i;

for (i=1; i<=m; i++)
    u2 *= ran3(n);

    return (-beta/m)*log(u2);
}

double normal(n, mean, stdev)
int n;
double mean;
double stdev;
{
    return (mean + gaussian(n) * stdev);
}

double lognormal(n, mean, stdev)
int n;
double mean;
double stdev;
{
    return (exp( normal(n, mean, stdev)));
}

double gaussian(n)
int n;
{
    static int iset = 0;
    static double gset;
double fac, f, v1, v2;

    if (iset == 0)
do {
    v1 = 2.0 * ran3(n) - 1.0;
    v2 = 2.0 * ran3(n) - 1.0;
    r = v1*v1 + v2*v2;
    j while (r >= 1.0);
    fac = sqrt(-2.0*Log(r)/r);
    gset = fac * v1;
    iset = 1;
    return (fac * v2);
} else {
    iset = 0;
    return gset;
}

/*
 * data collecting, statistical computation,
 * and reporting functions
 */

void mfisestat(n, entime)
int n; /* list number */
double entime; /* entry time */
{
    double faz1, dt;
    faz1 = (double) lb[n].lc;
    accumarea(faz1, &((lb[n].length), tnow);
    lb[n].lc = 1;
    dt = tnow - entime;
    notevall( dt, &((lb[n].wait )));
}

void pfisestat(n)
int n; /* list number */
{
    double faz1;
    faz1 = (double) lb[n].lc;
    accumarea(faz1, &((lb[n].length), tnow);
void cirtp(var)
  tpvar *var;
  {
    var->areaah = 0.0;
    var->heree = 0.0;
    var->ttot = tnow;
    var->tlast = tnow;
    var->tcr = tnow;
  }

void cirov(var)
  obsvar *var;
  {
    var->num = 0.0;
    var->sumx = 0.0;
    var->sumux = 0.0;
  }

void cirlist()
  {
    int faz;

    for (faz=0; faz<NILISTS; faz++)
      {
        lb[faz].lmax = 0;
        cirtp( &(lb[faz].length) );
        cirov( &(lb[faz].wait) );
      }
  }

void cirres()
  {
    int faz;

    for (faz=0; faz<NRES; faz++)
      {
        cirov( &(rn[faz].util) );
        rn[faz].idltimmax = 0;
      }
  }
rn[faz].bytimmmax = 0.;
rn[faz].entcount = 0;
}
)

void rrhead() {
    printf(" resource utilization ");
    printf(" maximum time\n");
    printf(" +---------------------+\n");
    printf(" +---------------------+\n");
    printf(" entity\n");
    printf(" num name cap cur ave stddev idle ");
    printf(" busy count\n");
    printf(" ----------------- ------------ ------------ ------------ ------------ ");
    printf(" -----------------\n");
}

void resource(num) {
    int num;
    {
        double faz, f1[2];
        faz = (double) rn[num].utilcur;
        getpsstat(f1, &rn[num].util1, faz);
        printf("%3d %10s %4d %4d %10.4e %10.4e %8g %8g %8d\n", 
            num, rn[num].name, rn[num].cap, rn[num].utilcur, f1[0], f1[1], 
            rn[num].idltimmmax, rn[num].bytimmmax, rn[num].entcount);
    }
}

void repres() {
    int faz;
    printf("\n\n\n");
    printf("RESOURCE STATISTICS: time = %12.5e\n
”, tnow);
    rrhead();
    for (faz=0; faz<NRES; faz++)
        resource(faz);
    printf("\n\n");
}
void replist() {
    int faz;
    printf("\n\n\n\n");
    printf("LIST STATISTICS: time = %12.5e\n\n\n", tnow);
    rhead();
    rlist(&evb, -1);

    for (faz=0; faz<NLISTS; faz++)
        rlist(&lb[faz], (az);

    printf("\n\n\n");
}

void rlist(a, n)
listnote *a;
int n;
{
    double faz1;
    double faz2[], khan[];

    faz1 = (double) a->lcur;
    getpeta(a, &a->length, faz1);
    getoveta(khan, &(a->wait));

    if (n >= 0)
        printf("%3d ", n);
    else
        printf(" ");

    printf("%10s %10.4e %10.4e %5d %5d %10.4e %10.4e \n", 
            a->name, faz[0], faz[1], a->lmax, a->lcur, 
            khan[0], khan[1]);
}

void rhead() {
    printf(" list length waiting\n");
    printf(" time\n");
    printf("----------------------------------- +--------\n");
    printf("--------\n");
    printf("num name ave stdev max cur ave ");
    printf(" stdev\n");
    printf("-------- ------- -------- ---- -------- \n");
}
printf("-------\n");
)

void getovstat(pvec, pov)
void gettpstat(pvec, ptp, value)

void notevalue(xobs, var)


void getovstat(pvec, pov)
double *pvec;
Observ *pov;
{
    pvec[0] = pov->sumv/pov->num;
    if (pov->num > 1.1)
        pvec[1] = sqrt((pov->sumv - pov->num * pvec[0] * pvec[0])
                       /(pov->num - 1.0));
    else
        pvec[1] = 0.0;
}

void gettpstat(pvec, ptp, value)
double *pvec;
double *ptp;

double  value;
{
    double dt;

    accumarea(value, ptp, tnow);

    if ( (dt = (ptp->ttot - ptp->ltot)) > 0.0 )
    {
        pvec[0] = ptp->areah/(dt);
        pvec[1] = sqrt((ptp->hharea/(dt)) - pvec[0] * pvec[0]);
    }
    else
        pvec[0] = pvec[1] = 0.0;
}

void notevalue(xobs, var)
double xobs;
Observ *var;
{
    var->num += 1.0;
    var->sumx += xobs;
    var->mean = var->sumx/var->num;
    var->sumx += xobs * xobs;
}
void accumareaheight, var, updtimer)
double height;
spvar *var;
dooble updtimer;
{
dooble dt, dt2;

var->tlast = var->ttot;
var->ttot = updtimer;
dt = var->ttot - var->tlast;
dt2 = var->ttot - var->tclr;

var->area += height * dt;
var->harea += height * height * dt;

if (dt2 > 0.0)
    var->mean = var->area/height/2;
}

double **getarray()
{
dooble **a, *b;
int i;

b = (double *) calloc( (50*(NODE + 2)), sizeof(double));
if (ib)
    perror("getarray(): unable to allocate space for array ");

a = (double **) calloc(50, sizeof(double *));
if (l)
    perror("getarray(): unable to allocate space for array ");

for (i=0; i<50; i++)
    {a[i] = b;
b += (NODE+2);
    }

return a;
}

int ib/4 = 0;
dooble **arrnum[20];

void binarr(pyb, time) /* fcn to collect values of continuous */
dooble *pyb; /* variables to be plotted later */
double time;
{
    int l, k;
    static int j = 0;
    static double **pdabld = NULL;

    if (!bld > (20*50-1))
        perror("blddir(): out of space for recording state variables");

    k = !bld % 50;

    if (k == 0)
    {
        arrnum[j] = getarray();
        pdabld = arrnum[j];
        j++;
    }

    for(i=0; i<NODE; i++)
    {
        pdabld[k][i] = pyb[i];
    }

    pdabld[k][NODE+1] = time;
    ibld++;
}

void cplot(cols)
int cols;
{
    int i, j, l, m, faz2;
    double k, temp, faz1, **pdabld;
    double hi[NODE], lo[NODE], wid[NODE],
            scaled[NODE], arr[NODE];
    int num(NODE);

    /* establish size of output */
    if (cols == 80)
    {
        faz1 = 50.;
        faz2 = 40;
    }
    else
    {
        faz1 = 100.;
        faz2 = 90;
    }

    /* initialize lo and hi values for */
    /* each state variable */
    for(i=0; i<NODE; i++)
lo[i] = hi[i] = arrnum[0][0][i];

/* find out actual lo and hi values */
/* for each s.v. */
for(m=0; m<ibld; m++)
{
    j = m % 50;
pdabl = arrnum[m/501];
for(i=0; i<NODE; i++)
{
    if(pdabl[j][i] > hi[i]) hi[i] = pdabl[j][i];
    if(pdabl[j][i] < lo[i]) lo[i] = pdabl[j][i];
}

/* - change lo and hi values if they are equal */
/* - establish widths of values for each s.v. */
/* - print out character representing each s.v., */
/* the lo value and the hi value */
for(i=0; i<NODE; i++)
{
    if (hi[i] == lo[i])
    {
        hi[i] = hi[i] + 0.5 * hi[i];
        lo[i] = lo[i] - 0.5 * lo[i];
    }
    width[i] = hi[i]-lo[i];
    printf("%c %d\n", i+97, lo[i]);
    for(l=0; l<fax2; l++)
    {
        printf(" ");
        printf("%d\n", hi[i]);
    }
}

/* print out times and plots of each s.v. */
for(m=0; m<ibld; m++)
{
    j = m % 50;
pdabl = arrnum[m/501];
for(i=0; i<NODE; i++)
    scaled[i] = (pdabl[j][i]-lo[i])/(hi[i]-lo[i]) * fax1;
for(i=0; i<NODE; i++)
{
    arr[i] = 1000.;
    for(l=0; l<NODE; l++)
    {
        if (arr[i] > scaled[l])
        {
            arr[i] = scaled[l];
            num[l] = 1;
        }
    }
}
scaled[num[i]] = 101.;

printf("%10.4e *", pdash[i][NODE+1]);
temp = 0.;
for(i=0; i<NODE; i++)
{
    for(k=1; k<(arr[i]-temp); k+=1.0)
        print(" ");
    printf("%8c, num[i]+97);  
    temp = arr[i];
}
printf("\n");
}

/*===========================================================================
 * error handling functions
 */

void checkt()
{
    if (tfull <= 0.0)
        perror("tdfull is not defined properly");

    if (tdrec <= 0.0)
        perror("tdrec is not defined properly");

    if (hmin <= 0.0)
        perror("hmin is not defined properly");

    if (hmax <= 0.0)
        perror("hmax is not defined properly");

    if (tolr <= 0.0)
        perror("tolr is not defined properly");

    if (tdfull > tdrec)
        perror("tdfull is larger than tdrec");

    if (hmin > hmax)
        perror("hmin is larger than hmax");
if (hmax > tdfull)
    perror("hmax is larger than tdfull");

if (tolr > hmin)
    perror("tolr is larger than hmin");

)

void perror(pstring)
char *pstring;
{
    /* function to print an error message and exit program */
    fprintf(stderr, "time = %f\n", tnow);
    fprintf(stderr, "%s \n", pstring);
    fprintf(stderr, "Aborting program.\n");
    abort();
    exit(1);
}

void perror1(pstring1, num)
char *pstring1;
int num;
{
    /* function to print an error message and exit program */

    fprintf(stderr, "time = %f\n", tnow);
    fprintf(stderr, "%s %d. \n", pstring1, num);
    fprintf(stderr, "Aborting program.\n");
    abort();
    exit(1);
}

void perror2(pstring1, num, pstring2)
char *pstring1;
int num;
char *pstring2;
{
    /* function to print an error message and exit program */

    fprintf(stderr, "time = %f\n", tnow);
    fprintf(stderr, "%s %d %s. \n", pstring1, num, pstring2);
    fprintf(stderr, "Aborting program.\n");
    abort();
    exit(1);
void advance(ta, tb)
    double ta, tb;
    {
        double ya[NODE], yb[NODE];
        boolean bool1;
        int i, faz1, faz2;
        int khan;
        double ttarg1, ttarg2;

        ttarg1 = ta;
        ttarg2 = tb;
        bool1 = 0;

        for (i=0; i < NODE; i++)
        {
            ya[i] = snow[i];
            slast[i] = ya[i];
            dlast[i] = dnow[i];
            yb[i] = 0.0;
        }

        do {
            desolve(ta, ttarg2, ya, yb);
            for (i=0; i<NODE; i++)
            {
                snow[i] = yb[i];
                yfirst(ttarg2, yb, dnow);
            }
        } while (NSTEVF)
        stevcond();
        faz1 = faz2 = 0;
        for (i=0; i<NSTEVF; i++)
        {
            khan = ABS(sflag[i]);
            if ( khan == 2)
            
            faz2++;
            if ( khan == 1)
            
            faz1++;}
        }
        if (bool1)
\begin{verbatim}
if (faz2)
{
    bool1++;
    ttarg2 = ttarg2 - (ttarg2 - ttarg1)/2.0;
}
else if (faz1 && faz2 == 0)
{
    tnow = ttarg2;
    update(yb, ttarg2);
    sevents();
    return;
}
else
{
    bool1++;  
    ttarg2 = ttarg2 + (ttarg2 - ttarg1)/2.0;
    ttarg1 = ttarg2;
}
else
{
    if (faz2)
    {
        bool1++;  
        ttarg2 = ttarg2 - (ttarg2 - ttarg1)/2.0;
    }
    else if (faz1 && faz2 == 0)
    {
        tnow = ttarg2;
        update(yb, ttarg2);
        sevents();
        return;
    }
    else
    {
        tnow = ttarg2;
        update(yb, ttarg2);
        return;
    }
}
}

boolean descive(a, b, *pya, *pyb)

double a, b, *pya, *pyb;
\end{verbatim}
double hold, x, yold[NODE], gamma,
    ynew[NODE], hnew, *pest;
boolean bool1;
int i;

hold = hmax;
x = a;
for(i=0; i<NODE; i++)
    yold[i] = y[i];

bool1 = 0;
do
{
    pest = rkf(x, yold, hold, ynew);
    gamma = gamma(x, b, yold, pest);
    hnew = 0.8 * gamma * hold;

    if ( gamma >= 1.0 )
    {
        if ( hnew < hmin )
        {
            perror("desolve(): step size needed was smaller than hmin");
        }
        else /* hnew >= hmin */
        {
            if ( hnew > 5.*hold ) hnew = 5.*hold;
            if ( hnew > hmax ) hnew = hmax;
            if ( x + hold >= b )
                bool1 = 1;
        }
        else
        {
            x = x + hold;
            hold = hnew;
            for(i=0; i<NODE; i++)
                yold[i] = ynew[i];
            bool1 = 0;
        }
    }
    else
    {
        if ( hnew < hold/10. ) hnew = hold/10.;
        if ( hnew < hmin )
        {
            perror("desolve(): step size needed was smaller than hmin");
        }
        else
        {
            hold = hnew;
            bool1 = 0;
        }
    }
}
void update(pyb, b)  
{  
    double *pyb, b;  
    int i;  
    for (i=0; i<nODE; i++)  
        snow[i] = pyb[i];  
    yfirst(b, pyb, snow);  
}

int cross(cgl, cgn, cdl, cdn, dir, tol)  
{  
    double diff;  
    /* cgl: crossing var. at tlast */  
    /* cgn: crossing var. at tnow */  
    /* cdl: crossed var. at tlast */  
    /* cdn: crossed var. at tnow */  
    /* dir: direction of crossing searching for */  
    /* tol: tolerance allowed in accepting crossing */  
    /* is cdl greater than, less than, or equal to cgl? */  
    /* cdl greater than cgl */  
    if (cdl > cgl)  
    {  
        if (cgn >= cdn)  
        {  
            if (dir > 0)  
            {  
                /* is the crossing within tolerance? */  
                diff = ABS(cgn-cdn);  
                if (diff <= tol)  
                    return 1; /* within tolerance */  
            }  
        }  
    }  
}
else
    return 2; /* not within tol. */
}
else /* dir is negative */
    return 0;
else /* cg is less than cdn */
    return 0;

/* cd is less than cg */
else if ( cd < cg )
{
    if ( cg <= cdn )
    {
        if ( dir <= 0 )
        {
            /* is the crossing within tolerance? */
            diff = ABS(cg-cdn);
            if ( diff <= tol )
                return 1; /* within tolerance */
            else
                return 2; /* not within tol. */
        }
        else /* dir is positive */
            return 0;
    }
    else /* cg is more than cdn */
        return 0;
}

/* cd is equal to cg */
else
    return 0;


double gamxsl(a, b, pycld, pest) 
double a, b, *pycld, *pest;
{
    double absbest, gamma, dump;
    int i;
    double gam2 = 1.610;

    for ( i = 0 ; i < NODE ; i++ )
    {
        absbest = ABS(pest[i]);
        if ( absbest == 0.0 )
gamma = 6.25;
else if ( pyold[i] == 0.0 )
    gamma = sqrt(sqrt(tolr * tolr / (absest * (b-s))));
else
    gamma = sqrt(sqrt(tolr * ABS(pyold[i])/(absest*(b-s))));

if ( gamma < gam2 )
    gam2 = gamma;
}

return gam2;


double *rkf(x, pyold, h, *pynw);
double x, *pyold, h, *pynw;
{
    double k1[NODE], k2[NODE], k3[NODE],
        k4[NODE], k5[NODE], k6[NODE];
    double b21 = 1./4.,
        b31 = 3./32., b32 = 9./32.,
        b41 = 1932./2197., b42 = -7200./2197.,
        b43 = 7296./2197.,
        b51 = 439./216., b52 = -8.,
        b53 = 3680./513., b54 = -845./514.,
        b61 = -8./27., b62 = 2., b63 = -3544./2565.,
        b64 = 1859./4104., b65 = -11./40.;
    double a1 = 25./216., a3 = 1408./2565.,
        a4 = 2197./4104., a5 = 1./5.;
    double r1 = 1./360., r3 = -355./4752.,
        r4 = -4752./50176., r5 = 1./25., r6 = 2./55.;
    double yt[NODE], xt, est[NODE];
    int i;

    for ( i=0 ; i< NODE ; i++ )
        yt[i] = pyold[i];
    xt = x;
    yfirst(xt, yt, k1);

    for ( i=0 ; i< NODE ; i++ )
        yt[i] = pyold[i] + h*b21*k1[i];
    xt = x + h/4.0;
    yfirst(xt, yt, k2);

    for ( i=0 ; i< NODE ; i++ )
        yt[i] = pyold[i] + h*(b31*k1[i] + b32*k2[i]);
    xt = x + 3.*h/8.;
    yfirst(xt, yt, k3);

    for ( i=0 ; i< NODE ; i++ )
yt[i] = pyold[i] + h*(b41*k1[i] + b42*k2[i] + b43*k3[i]);
x = x + 12.*h/13.;
yfirst(xt, yt, k4);

for ( i=0 ; i<NODE ; i++ )
    yt[i] = pyold[i] + h*(b51*k1[i] + b52*k2[i] + b53*k3[i] + b54*k4[i]);
x = x + h;
yfirst(xt, yt, k5);

for ( i=0 ; i<NODE ; i++ )
    yt[i] = pyold[i] + h*(b61*k1[i] + b62*k2[i] + b63*k3[i]
                         + b64*k4[i] + b65*k5[i]);
x = x + h/2.;
yfirst(xt, yt, k6);

for ( i=0 ; i<NODE ; i++ )
{
    pynew[i] = pyold[i] + h*(a1*k1[i] + a3*k3[i] + a4*k4[i] + a5*k5[i]);
    est[i] = r1*k1[i] + r3*k3[i] + r4*k4[i] + r5*k5[i] + r6*k6[i];
}
return est;

double whennext() /* returns the time of the next time event */
{
    return evlist->evtime;
}

/*****************************/
Appendix B

Listing of CBST model programs
Appendix B.1a

Listing of M/M/1 queue model

/*
 * M/M/1 queueing model using CBST
 *
*/

/*
 * The user should #include the math.h header
 * and the stdio.h header. In this case,
 * "vanwyk.h" #includes stdio.h.
 */
#include "vanwyk.h"
#include <math.h>

/*
 * The user must #define the constants for:
 * - number of lists
 * - number of resources
 * - number of random number streams
 * - number of double in array r[1] of the
 *   entity declaration
 *
 * The minimum that each may have is 1.
 */
#define NLISTS 1
#define NRES 1
#define NSEEDEDS 1
#define NR 1

/*
 * The user must make the entity structure definition
 */
typedef struct entity entity;
struct entity {
    double r[NR];
};
*/
* The user must then declare the functions which
* are written in this file
*
void unit(), tevent();
*/

#include "xdisc.h"
#include "xcbst.h"

/*
* The user must then #include the proper header file
* containing the functions required for combined
* simulation
*/

double iatime, servtime;

/*
* This model uses the global variables:
* - iatime, the interarrival time
* - servtime, the average service time
*/

main()
{
    iatime = 10.;
    servtime = 9.;

    discrete();
    exit(1);
}
/ * It is desired to stop the simulation run 
 * after 5000 minutes of simulation time. 
 */

int stopcheck()
{
    if(tnow >= 5000.)
        return 1;
    else
        return 0;
}

/*
 * in uinit();
 * - set the capacity of the resource
 * - name the list and the resource
 * - schedule some time-events
 */

void uinit()
{
    entity *slf;

    rl[0].cap = 1;
    lb[0].name = "queue 0";
    rl[0].name = "server 0";

    /* schedule the first arrival at time 0 */
    slf = makeentity();
    schedule(1, 0.0, slf);
    schedule(4, 5000., NULL);
}

/*/ 
 * tevents() must be written to input a variable 
 * of type (element *). This is used to access 
 * the event type from the element. Control is 
 * sent to the proper time-event function using
* this data. The format used below is highly
* recommended.
*/

void tevents(a)
  element *a;
{
  void arrival(), endserve(), report();

  if (a->evtype == 1)
    arrival(a);
  else if (a->evtype == 2)
    endserve(a);
  else if (a->evtype == 4)
    report();
  else
    perror("Errant event code", 1);

  free(a);
}

/*
 * report() is event type 4. It is used to
 * generate a report.
 */

void report()
{
  replist();
  repres();
}

/*
 * arrival() goes through the logic associated
 * with the event of an arrival of an entity
 * into the system.
 */

void arrival(b)
  element *b;
{
  double x;
  entity *a;

  /* schedule next arrival */
a = makeentity();
schedule(1, x = tnow + exp(0, latime), a);

/* check if any servers are available */
if ( resavail(0) )
{
    /* if they are avail, then schedule end of serv */
    notevalue(0.0, &lb[0].wait);
    rescapture(0, 1);
    schedule(2, x = tnow + exp(0, servtime), b->pent);
}
else
{
    /* if servers are busy, put customer in queue */
    filefifo(0, b->pent);
}

free(b);

/*
 * endserve() goes through the logic for the
 * time-event of ending of service for an entity.
 * This is also when an entity leaves the system.
 */
void endserve(a)
element *a;
{
    double x;
    entity *b;

    resrelease(0, 1);

    /* check if num in queue > 0 */
    if ( lb[0].lcur > 0 )
    {
        /* if num in queue > 0, then remove next cust */
        /* and schedule end of service for him */
        b = remove1st(0);
        schedule(2, x = tnow + exp(0, servtime), b);
        rescapture(0, 1);
    }

    free(a->pent);
    free(a);
}
Appendix B.1b

Rewritten main() function

```c
main()
{
    int i, j;
    static int sseed[10] = { -111, -345, -2458, -1009823, -1, -9090912,
                           -5682416, -7251935, -187354, -90812345 };
    static char *str[3] = { "util", "length", "wait" };
    double rho, len, faz[2], f;
    observe khan[3];

    for (i=0; i<3; i++)
        clrow( &khan[i] );

    for (i=0; i<3; i++)
    {
        rho = serv[i]/iatime;
        len = (rho*rho/(1.0-rho));
        printf("\n\ninterarrival time = 10.0, service time = %3.1f\n", serv[i]);
        printf("expected:  u = %6.4f l = %8.4f w = %8.4f\n", rho, len, iatime*len);
        servtime = serv[i];
        for (j=0; j<10; j++)
        {
            printf("\nRun %2d: ", j);
            seed[0] = sseed[j];
            discrete();
            printf("u = %8.4f l = %11.4f w = %11.4f\n",
                   rn[0].util.mean, lb[0].length.mean, lb[0].wait.mean);
            notevalue( rn[0].util.mean, &khan[0] );
            notevalue( lb[0].length.mean, &khan[1] );
            notevalue( lb[0].wait.mean, &khan[2] );
        }

        printf("\n\n95 percent confidence intervals\n");
        for (j=0; j<3; j++)
        {
            getovstat(faz, &khan[j] );
            f = 2.26 * (faz[1]/sqrt(10));
            printf("\n\n%8.4f %8.4f %8.4f\n",
                   str[j], faz[0], f, (faz[0]-f), (faz[0]+f) );
            clrow( &khan[j] );
        }
    }
    exit(1);
}
```
Appendix B.2

Listing of Cedar Bog Lake model

/* cedar bog lake: continuous simulation using CBST */

/*
  * The user should #include the math.h header
  * and the stdio.h header. In this case,
  * "varwyk.h" #includes stdio.h.
  */

#include "varwyk.h"
#include <math.h>

/*
  * The user must #define the constants for:
  *   - number of state variables
  */

/#define NODE 5
/#define NSTEVF 1

/*-----------------------------*/

/*
  * The user must then declare the functions which
  * are written in this file
  */

void sevents(), stevcond(), yfirst(), unit();
/*
 * The user must then #include the proper header file
 * containing the functions required for combined
 * simulation and the special header file: "xcont.h"
 * for continuous simulation.
 */

#include "xcont.h"
#include "xcbrt.h"

/*
 * stevcond() is used to check for state-events.
 * It is called by advance(). It sets the sflags
 * based upon whether a state variable crosses a
 * threshold. In this case, there are no state-events
 * and thus, no state-event conditions. Both stevcond()
 * and sevents() are thus empty.
 */

void stevcond()
{

}

void sevents()
{

}

/*
 * The modeler uses main() to start the
 * simulation by calling continuous()
 */

main()
{
    continuous();
    exit(1);
}

/*
 * yfirst() is the function in which the rate of
 * change of the state variables is coded.
*/
void yfirst(t, y, ddy)
    double t, *y, *ddy;
{
    double temp;

    temp = 95.9 * (1.0 + 0.635 * sin(2.0 * PI * t));
    ddy[0] = temp - 4.03 * y[0];
    ddy[1] = 0.48 * y[0] - 17.87 * y[1];
    ddy[6] = y[0] + 6.9 * y[1] + 2.7 * y[2];
}

/*
 * It is desired to stop the simulation
 * after 2 minutes of simulation time
 * and to plot the state variable values
 */

int stopcheck()
{
    if(tnow > 2.)
    {
        cplot(80);
        return 1;
    }
    else
    {
        return 0;
    }
}

/*
 * in uinit():
 * - initialize some CST global variables
 * - set initial values of state variable
 */

void uinit()
{
    tdfull = 0.005;
}
\text{tdrec} = 5.0*\text{tdfull};
\text{hmin} = 0.01*\text{tdfull};
\text{hmax} = \text{tdfull};
\text{tolr} = 6.001*\text{tdfull};

\text{s1last[0]} = \text{snow[0]} = 0.83;
\text{s1last[1]} = \text{snow[1]} = .003;
\text{s1last[2]} = \text{snow[2]} = .0001;
\text{s1last[3]} = \text{snow[3]} = 0.0;
\text{s1last[4]} = \text{snow[4]} = 0.0;

\)}
Appendix B.3

Listing of Tank/truck model

/*
 *  The combined simulation model of
 *  the tank/truck system
 */

/*
 *  The user should #include the math.h header
 *  and the stdio.h header. In this case,
 *  "vanwyk.h" #includes stdio.h.
 */

#include "vanwyk.h"
#include <math.h>

/*
 *  The user must #define the constants for:
 *  - number of state variables
 *  - number of state event functions
 *  - number of lists
 *  - number of resources
 *  - number of random number streams
 *
 *  The minimum that each may have is 1.
 */

#define NODE 1
#define NSTEVF 4
#define NLISTS 1
#define NRES 1
#define NSEEDS 1
#define NR 1

typedef struct entity entity;
struct entity {
    int trucknum;
}


```c
double r[NR];
);

/
* The user must then declare the functions which
* are written in this file
*/

void sevets(), stevcond(), yfirst(), uinit(), tevents();

/
* The user must then #include the proper header file
* containing the functions required for combined
* simulation
*/

#include "xcbst.h"

/
* This model requires the two global integers:
* fbay and fvalue. These are flags indicating whether
* the loading bay and the outlet valve
* are open (1) or closed (0)
*/

int fbay = 1;
int fvalue = 0;

/
* The modeler uses main() to start the
* simulation by calling combined()
*/

main()
{
    combined();
    exit(1);
}

/*
It is desired to stop the simulation after 53 minutes of simulation time

```c
int stopcheck()
{
    if (tnow > 53.)
        return 1;
    else
        return 0;
}
```

/*
 * in uinit():
 * - change some of the CBST default values
 * - schedule some time-events
 * - initialize some CBST global variables
 * - set initial values of state variable
 */

```c
void uinit()
{
    entity *slf;

    /* set capacity of resource 0 to 1 unit */
    r0().cap = 1;

    /* name the zeroth list and resource */
    lb[0].name = "q_lbay";
    r0().name = "lbay";

    /* schedule the three arrivals at times */
    /* 5.0, 12.5, and 14.0 minutes */
    slf = makeentity();
    slf->trgnum = 1;
    schedule(1, 5.0, slf);

    slf = makeentity();
    slf->trgnum = 2;
    schedule(1, 12.5, slf);

    slf = makeentity();
    slf->trgnum = 3;
    schedule(1, 14.0, slf);

    /* schedule an event of type 3 -- report */
    slf = makeentity();
    schedule(3, 53., slf);
```
/* set the global variables: tdfull, tddrec */
/* hmin, hmax, and tolr */
tdfull = 0.1;
tdtddrec = 5.0*tdfull;
hmin = 0.01*tdfull;
hmax = tdfull;
tolr = 0.001*tdfull;

/* set the initial values of the state variable */
slast[0] = 65.0;
snow[0] = 65.0;
}

/*
 * stevcond() is used to check for state-events.
 * It is called by advance(). It sets the sflags
 * based upon whether a state variable crosses a
 * threshold.
 */
void stevcond()
{
    sflag[0] = cross(slast[0], snow[0], 95.0, 95.0, 1, .001);
    sflag[1] = cross(slast[0], snow[0], 100.0, 100.0, 1, .001);
    sflag[2] = cross(slast[0], snow[0], 95.0, 95.0, -1, .001);
    sflag[3] = cross(slast[0], snow[0], 70.0, 70.0, -1, .001);
}

/*
 * yfirst() is the function in which the rate of
 * change of the state variables is coded. The code
 * below corresponds to the equation:
 * 
 * \[ \frac{dy}{dt} = 2 * r[n].utilcur - 2 * fvalve \]
 * 
 * where: \( r[n].utilcur \) is 1, if resource 0, the loading bay,
 *        \( r[n].utilcur \) is currently being used, and 0, if not.
 * 
 * and \( fvalve \) is 1, if the valve is open
 * or 0, if it is closed
 */
void yfirst(t, py, pk)
double t, *py, *pk;

178
\( p[0] = 2.0 \cdot \text{rtime} - 2.0 \cdot fval; \)

/*
 * sevents() sends control to the proper state-event
 * function(s) depending on the vector of state-event
 * flags -- sflag[]. Note the difference with sevents().
 * Since more than 1 state-event may occur at the same
 * time, the if-else control structure is not used.
 * sevents() is called by advance().
*/

void sevents()
{
    void closebay(), openvalve(), openbay(), closevalve();
    int i;

    if (sflag[0] == 1)
        closebay();
    if (sflag[1] == 1)
        openvalve();
    if (sflag[2] == 1)
        openbay();
    if (sflag[3] == 1)
        closevalve();
}

void closebay()
{
  /* close the bay */
  printf("time = %10.3f, level = %10.3f: close the bay\n",
         tnow, snow[0]);
  fbay = 0;
}

void openbay()
{
    entity *b;

    /* open the bay */
    printf("time = %10.3f, level = %10.3f: open the bay\n",
           tnow, snow[0]);
    fbay = 1;

    /* The bay is open. If queue 0 is non-empty, */
    /* then schedule an end of service for the */
/* next truck */
if (lb[0].icur)
{
    b = remove1st(0);
    schedule(2, (tnow + 10.0), b);
    recapture(0, 1);
}

void closevalve()
{
    fvalve = 0;    /* closed */
    printf("time = %10.3f, level = %10.3f: close the valve\n", tnow, snow[0]);
}

void openvalve()
{
    fvalve = 1;    /* open*/
    printf("time = %10.3f, level = %10.3f: open the valve\n", tnow, snow[0]);
}

/*
 * tevents() must be written to input a variable
 * of type (element *). This is used to access
 * the event type from the element. Control is
 * sent to the proper time-event function using
 * this data. The format used below is highly
 * recommended.
 */

void tevents(a)
element *a;
{
    void report(), arrival(), endserve();

    if (a->evtype == 1)
        arrival(a);
    else if (a->evtype == 2)
        endserve(a);
    else if (a->evtype == 3)
        report();
    else
        perror("Errant event code", 1);
}
void report()
{
    replist();
    repres();
    cplot(80);
}

void arrival(truck)
{element *truck;

double x;

    truck->pent->r[0] = tnow;

    printf("time = %10.3f, level = %10.3f: arrival of truck %2d
", 
        tnow, smow[0], truck->pent->trucknum);

    /* check if bay is open */
    if (fbay

        /* check if queue is empty */
        if (lb[0].lcur == 0 && rn[0].utilcur == 0)
            
                notevalue(0.0, &lb[0].wait);
            rescapture(0, 1);
            schedule(2, x = tnow + 10.0, truck->pent);
        }
    else
        
            /* if bay is busy, put truck into queue */
            filefifo(0, truck->pent);
    }
}

free(truck);
/*
 * endserve() is the time-event function for the
 * logic associated with the end of service for
 * a truck.
 */

void endserve(truck)
    element *truck;
{
    double x;
    entity *b;

    tnow = truck->evtime;
    resrelease(0, 1);

    printf("time = %10.3f, level = %10.3f: end of service for truck %2d\n",
            tnow, snow[0], truck->pent.trucknum);

    /* If queue 0 is non-empty and bay is open, */
    /* then schedule an end of service for the */
    /* next truck */
    if (lb[0].lcurs && fbay )
    {
        b = remove1st(0);
        schedule(2, x = tnow + 10.0, b);
        rescapture(0, 1);
    }

    free(truck);
}
Appendix B.4

Listing of CIM Lab model

/* CIM Lab model */

/*
 * The user should #include the math.h header
 * and the stdio.h header. In this case,
 * "vanwyk.h" #includes stdio.h.
 */

#include "vanwyk.h"
#include <math.h>

/*
 * The user must #define the constants for:
 * - number of lists
 * - number of resources
 * - number of random number streams
 * -
 * The minimum that each may have is 1.
 */

#define NLISTS 9
#define NRES 10
#define NSEEDS 1
#define NR 1

typedef struct entity entity;

struct entity {
  int type;
  int dest;
  int p[5];
  double r[NR+1];
};

/*
 * Within the entity definition, the variables
 * represent the following:
type - the pallet type; can be 1, 2, 3, 4, or 5

dest - the current destination of the pallet;
    can be 0, 1, 3, 6, or 9

p[0] - # of cnc blanks

p[1] - # of robot blanks

p[2] - # of link blanks

p[3] - # of machined cnc bases

p[4] - # of machined cnc links

p[5] - # of machined robot bases

p[6] - # of machined robot links

p[7] - # of finished cnc's

p[8] - # of finished robots

*/

/* The user must then declare the functions which
   are written in this file
*/

void unit(), tevent();

/*
 * The user must then #include the proper header file
 * containing the functions required for discrete
 * simulation
 */
#include "xdisc.h"
#include "xcbst.h"

/*
 * define global variables to be used
* In the functions below
*/

/* all times are in minutes */
double t2 = 1.0,
    t14 = (7.0/60.0),
    t52 = (7.0/60.0),
    t92 = (7.0/60.0),
    t17 = (27.0/60.0),
    t82 = (27.0/60.0),
    t47 = (20.0/60.0),
    t85 = (20.0/60.0),
    t80 = (20.0/60.0),
    t34 = (7.0/60.0),
    t43 = (7.0/60.0),
    t67 = (7.0/60.0),
    t76 = (7.0/60.0),
    t45 = (3.0/60.0),
    t54 = (3.0/60.0),
    t78 = (3.0/60.0),
    t66 = 1.75, /* assemble robot */
    t77 = 0.75, /* build kit robot */
    t64 = 1.25, /* assemble cnc */
    t65 = 0.50, /* build kit cnc */
    tm1 = 5.0, /* machine robot */
    tm2 = 5.0; /* machine cnc */

int feeder1, feederr, feederc;
int pins1, pins2, pins3, pins4, pins5;
int cagg, ragg, cfin, rfin, occr, rcur;

void decmker();
extent *findpl();
void dpl();
void initpl();
void showlist();

/*
 * The modeler uses main() to start the
 * simulation by calling discrete()
 */

main()
{
    discrete();
    exit(1);
}
void showlist(q)

element *q;
{
    int i = 1;

    while(q && (i < 5)) /* take out i<5 req. */
        {
        printf("\n%2d evtype = %2d evtime = %f\n", i, q->evtype, q->evtime);
            if (q->pent)
                dpal(q->pent);
            q = q->after;
            i++;
        }
    
    void dpal(p)
    entity *p;
    {
        int i;

        printf("pallet %u ", p);
        printf(" type = %d ", p->type);
        printf(" dest = %d ", p->dest);
        for (i=0; i<9; i++)
            if (p->p[i])
                printf(" p[%d] = %2d", i, p->p[i]);
        printf("\n");
        printf("\n");
    }

    void unit()
    {
        int i;
        entity *order;
        entity *pallet;

        /* initialize feeders and pallets in */
        /* system counters. */
        feederl = feederc = feederr = 0;
        pins1 = pins2 = pins3 = pins4 = pins5 = 0;

        /* name the resources and lists */
        rnl[0].name = "crane";
        rnl[1].name = "crane_res";
        rnl[2].name = "asmb";
        rnl[3].name = "asmb_res";

        186
rn[4].name = "w/c";
rn[5].name = "w/c_res";
rn[6].name = "conv";
rn[7].name = "conv_res";
rn[8].name = "oper";
rn[9].name = "oper_res";

/* set capacities of resources */
for (i=0; i<NRES; i++)
    rni[i].cap = 1;

lb[0].name = "order_wait";
lb[1].name = "order_done";
lb[2].name = "wt_asmh";
lb[3].name = "wt_mach";
lb[4].name = "asrs_p0";
lb[5].name = "asrs_p1";
lb[6].name = "asrs_p2";
lb[7].name = "asrs_p3";
lb[8].name = "asrs_p4";

/* schedule a report */
schedule(35, 480.0, NULL);
schedule(35, 4800.0, NULL);

/* schedule first order */
order = makeentity();
order->p[7] = poisson(0, 10);
order->p[8] = poisson(0, 10);
schedule(0, 0.0, order);

/* initialize the as/rs */
for (i=0; i<7; i++)
{
    pallet = makeentity();
    initp(pallet);
    pallet->type = 1;
    pallet->p[0] = 9;
    filefifo(4, pallet);
}

for (i=0; i<5; i++)
{
    pallet = makeentity();
    initp(pallet);
    pallet->type = 2;
    pallet->p[1] = 12;
    filefifo(5, pallet);
}

for (i=0; i<8; i++)

(  pallet = makeentity();
  initpal(pallet);
  pallet->type = 3;
  pallet->p[21] = 21;
  filefifo(6, pallet);
)

for (i=0; i<40; i++)
{
  pallet = makeentity();
  initpal(pallet);
  pallet->type = 4;
  filefifo(7, pallet);
}

for (i=0; i<40; i++)
{
  pallet = makeentity();
  initpal(pallet);
  pallet->type = 5;
  filefifo(8, pallet);
}

void initpal(p)
entity *p;
{
  int i, j;

  for (i=0; i<9; i++)
    p->p[i] = 0;

  p->dest = 1;

  p->type = 0;
}

/*
 * It is desired to stop the simulation
 * after 4800 minutes of simulation time
 */

int stopcheck()
{
if (snov > 4800.)
    return 1;
else
    return 0;
}

void tevents(a)
    element *e;
{
    void order(), mv01(), mv14(), mv17(), mv20(), mv34(),
        mv43(), mv45(), mv47(), mv52(), mv54(), mv67(),
        mv76(), mv78(), mv82(), mv85(), mv89(), mv92(),
        mv95(), asbreak(), mcbreak(), endmc(), endass(),
        endoper(), report();

    if (a->evtype == 0)
        order(a);
    else if (a->evtype == 1 )
        mv01(a);
    else if (a->evtype == 2 )
        mv14(a);
    else if (a->evtype == 3 )
        mv17(a);
    else if (a->evtype == 4 )
        mv20(a);
    else if (a->evtype == 5 )
        mv34(a);
    else if (a->evtype == 6 )
        mv43(a);
    else if (a->evtype == 7 )
        mv45(a);
    else if (a->evtype == 8 )
        mv47(a);
    else if (a->evtype == 9 )
        mv52(a);
    else if (a->evtype == 10 )
        mv54(a);
    else if (a->evtype == 11 )
        mv67(a);
    else if (a->evtype == 12 )
        mv76(a);
    else if (a->evtype == 13 )
        mv78(a);
    else if (a->evtype == 14 )
        mv82(a);
    else if (a->evtype == 15 )
        mv85(a);
    else if (a->evtype == 16 )
        mv89(a);
    else if (a->evtype == 17 )
mv92(a);
else if (a->evtype == 18 )
   mv95(a);
else if (a->evtype == 19 )
   assbreak(a);
else if (a->evtype == 20 )
   mbbreak(a);
else if (a->evtype == 21 )
   endcm(a);
else if (a->evtype == 22 )
   endass(a);
else if (a->evtype == 23 )
   endoper(a);
else if (a->evtype == 35 )
   report();
else
   perror1("tevents: unknown event type: ", a->evtype);
}

void assbreak()
void mbbreak()
void endoper()

void report()
{
   printf("\n\n\n\time = %f\n\n", tnow);
   printf("cagg = %4d  ragg = %4d  ccur = %2d  rcur = %2d  cfin = %2d  rfin = %2d\n\n", cagg, ragg, ccur, rcur, cfin, rfin);
   replist();
   repres();
}

void order(a)
element *a;
{
   /* 0 */

   entity *b;
   printf("\n\n\n\order(), time = %f\n", tnow);
   dpal(a->pent);

   printf("cagg = %4d  ragg = %4d  ccur = %2d  rcur = %2d  cfin = %2d  rfin = %2d\n\n", cagg, ragg, ccur, rcur, cfin, rfin);
/** schedule next order */
b = makenenty();
b->p[7] = poisson(0, 10);
b->p[8] = poisson(0, 10);
schedule(0, tnow + expom(0, 200.), b);

/** update aggregate order counters */
cagg += a->pent->p[7];
regg += a->pent->p[8];

/** update current order counters if there are no unfulfilled orders waiting */
if ( lb[0] . lcur == 0 )
{
  ccur = a->pent->p[7];
  rcur = a->pent->p[8];
}

/** place the order in the unfulfilled orders file */
filefifo(0, a->pent);

/** call decision maker */
decmaker(0, NULL);
}

tntity *findp(i, j)
int i; /* pallet type */
int j; /* extra info */
{
  entity *temp;
element *p;
  int fnum, faz;

  fnum = i + 3;

  p = list(fnum);
  if (!p)
    return NULL;

  if ( j < -1 )
    {
    while ( p->after & ! p->pent->p[j] )
    p = p->after;
    if ( ! p->pent->p[j] )
    return NULL;
    else
      {
    fixlist(fnum, p);

  191
mfilestat(fnum, p->tentry);

temp = p->pent;
free(p);
return temp;
}
}

else /* j is negative (looking for an empty) */
{
  faz = 0;
  for(i=0; i<9; i++)
    faz += p->pent->p[i];

  while (faz != 0 )
  {
    if ( p->after)
      p = p->after;
    else
      return NULL;
    faz = 0;
    for(i=0; i<9; i++)
      faz += p->pent->p[i];
  }

  if ( faz)
    return NULL;
else
  {
    fixlist(fnum, p);
    mfilestat(fnum, p->tentry);

    temp = p->pent;
    free(p);
    return temp;
  }
}


void mv01(a)
  element *a;
{
  /* 1 */

  entity *pallet;
  pallet = a->pent;
if ( pallet->dest == 3 )
    schedule(2, tnow + t14, pallet);
else if ( (pallet->dest == 6) || (pallet->dest == 9) )
    schedule(3, tnow + t17, pallet);
else
    perror("mv01(): errant destination");

    /* release crane and make it available */
    resrelease(0, 1);

    /* capture the conveyor */
    rescapture(6, 1);

    /* call decision maker */
    decmaker(1, NULL);

void mv14(a)
    element *a;
{
    /* 2 */

    entity *pallet;
    pallet = a->pent;

    /* release the conveyor */
    resrelease(6, 1);
    resrelease(7, 1);

    /* schedule end of move from 4 to 3 */
    schedule(6, tnow + t63, pallet);

    /* call decision maker */
    decmaker(2, NULL);
}

void mv17(a)
    element *a;
{
    /* 3 */

    entity *pallet;
    pallet = a->pent;
if (pallet->dest == 6)
{
    release(6, 1);
    release(7, 1);
    schedule(12, tnow + t76, pallet);
}
else if (pallet->dest == 9)
    schedule(13, tnow + t76, pallet);
else
    perror("mv17(): errant destination", pallet->dest);

/* call decision maker */
decmaker(3, NULL);
}

void mv20(s)
    element *a;
{
    /* 4 */
    entity *cure;
    entity *pallet;
    pallet = a->pent;

    /* release actual crane and crane op. */
    release(0, 1);

    /* put pallet back onto the as/rs */
    filefifo(3 + pallet->type, pallet);

    /*
    printf("\n-------- list %d\n", (3 + pallet->type));
    showlist(list[3 + pallet->type]);
    */

    /* update orders stats if need be */
    if (pallet->p[7])
        cfin++;
    if (pallet->p[8])
        rfin++;
    if (cfin >= cc + & rfin >= rcur)
    {
        /* there are enough finished parts */
        /* to fulfill an order */
        cfin = cc;
        rfin = rcur;
    }
carg = ccur;
ragr = rcur;
cure = remove1st(0);
filefifo(1, cure);
if ( !empty(list[0]) )
{
    carg = list[0]->pent->p[7];
    ragr = list[0]->pent->p[8];
}

decmaker(4, NULL);

/* call decision maker */

/* m34(a) */
element *a;
{
    /* 5 */
    entity *pallet;
pallet = a->pent;

    /* release the assembly system */
    resrelease(3, 1);

    /* capture the conveyor */
    rescapture(6, 1);

    /* schedule next move */
    if ( pallet->dest == 9 )
        schedule(8, tnow + t47, pallet);
    else if ( (resavail(5)) && ( ((pallet->type == 4) && (pallet->p[0])) ||
                ((pallet->type == 5) && (pallet->p[1]))) )
        {
            rescapture(5, 1); /* reserve m/c system */
pallet->dest = 6;
schedule(8, tnow + t47, pallet);
        }
    else
        {
            pallet->dest = 0;
schedule(7, tnow + t45, pallet);
        }

    /* call decision maker */
decmaker(5, NULL);

void mv43(a)
   element *a;
{
   /* 6 */
   int faz1, faz2;
   double ta1, ta2, ta3;

   entity *pallet;
   pallet = a->pent;

   /* capture the assembly operation */
   rescapture(2, 1);

   /* schedule end of assembly based upon */
   /* pallet type and contents */
   if ( (pallet->type == 1) ) /* cnc blanks */
   {
      ta1 = 0.0;
      faz1 = pallet->p[0];
      faz2 = feederc;
      while ( (faz1) && (faz2 < 6) )
      {
         faz1--;
         faz2++;
         ta1 += 0.25;
      }
      schedule(22, tnow + ta1, pallet);
   }
   else if ( (pallet->type == 2) ) /* robot blanks */
   {
      ta2 = 0.0;
      faz1 = pallet->p[1];
      faz2 = feederr;
      while ( (faz1) && (faz2 < 6) )
      {
         faz1--;
         faz2++;
         ta2 += 0.25;
      }
      schedule(22, tnow + ta2, pallet);
   }
   else if ( (pallet->type == 3) ) /* link blanks */
   {
      ta3 = 0.0;
      faz1 = pallet->p[2];
   
196
faz2 = feeder1;
while ((faz1) && (faz2 < 11))
{
    faz1--;
    faz2++;
    ta3 += 0.25;
}
schedule(22, tnow + ta3, pallet);
}
else if (pallet->type == 4) /* cnc kits */
{
    /* find out what is on pallet */
    if (pallet->p[4])
        /* machined parts are on pallet */
        schedule(22, tnow + ta4, pallet);
    else
        /* pallet is empty */
        schedule(22, tnow + ta5, pallet);
}
else if (pallet->type == 5) /* robot kits */
{
    /* find out what is on pallet */
    if (pallet->p[5])
        /* machined parts are on pallet */
        schedule(22, tnow + ta6, pallet);
    else
        /* pallet is empty */
        schedule(22, tnow + ta7, pallet);
}
else
    perror1("mv43(): unknown pallet type: ", pallet->type);

/* call decision maker */
decmaker(6, NULL);
}

void mv45(a)
element *a;
{
    /* 7 */
    entity *pallet;
pallet = a->pent;

    /* schedule a move from 5 to 2 */
schedule(9, tnow + ts2, pallet);

    /* call decision maker */
decmaker(7, NULL);
}

void mv47(a)
element *a;
{
    /* 8 */
    entity *pallet;
pallet = a->pent;
    /* note: this event can only take place */
    /* with a pallet coming out of the */
    /* assembly operation */
    /* release the conveyor */
    resrelease(6, 1);
    resrelease(7, 1);
    /* schedule end of move from 7 to 6 */
    schedule(12, tnow + t76, pallet);
    /* call decision maker */
    decmaker(8, NULL);
}

void mv52(a)
element *a;
{
    /* 9 */
    entity *pallet;
pallet = a->pent;
    /* capture the crane itself */
    if (resavail(0))
    {
        rescapture(0, 1);
        /* release the conveyor */
        resrelease(6, 1);
        resrelease(7, 1);
        /* schedule a storage */
        schedule(4, tnow + tairs, pallet);
    }
    else
    perror("mv52(): crane was not available");
}
/* call decision maker */
decmaker(9, NULL);
}

void mv54(a)
element *a;
{
    /* 10 */
    entity *pallet;
pallet = a->pent;

    /* release the conveyor */
    resrelease(6, 1);
    resrelease(7, 1);
    /* schedule end of move from 4 to 3 */
schedule(6, tnow + t43, pallet);

    /* call decision maker */
decmaker(10, NULL);
}

void mv67(a)
element *a;
{
    /* 11 */
    entity *pallet;
pallet = a->pent;

    /* release the machining system */
    resrelease(5, 1);

    /* capture the conveyor */
    rescapture(6, 1);

    /**/ if (resavail(3) && (pallet->p[3] || pallet->p[5])) {
        pallet->dest = 3;
        rescapture(3, 1);
    } else {
        pallet->dest = 0;
    }
/* schedule end of move from 7 to 8 */
schedule(13, tnow + t78, pallet);

/* call decision maker */
decmaker(11, NULL);
}
schedule(16, tnow + t89, pallet);
else
    perror("mv78(): Errant pallet destination: ", pallet->dest);

    /* call decision maker */
    decmaker(13, NULL);
}

void mv82(a)
    element *a;
{
    /* 14 */
    entity *pallet;
    pallet = a->pent;

    if (resavail(0))
    {
        /* release the conveyor */
        resrelease(6, 1);
        resrelease(7, 1);
        /* capture the crane itself */
        rescapture(0, 1);

        /* schedule a storage */
        schedule(4, tnow + t89, pallet);
    }
    else
        perror("mv82(): crane was not available");

    /* call decision maker */
    decmaker(14, NULL);
}

void mv85(a)
    element *a;
{
    /* 15 */
    entity *pallet;
    pallet = a->pent;
/* schedule an end of move from 5 to 4 */
if (pallet->dest == 3)
    schedule(10, now + t54, pallet);
else
    perror("mv85(): pallet dest was not 3, it was: ", pallet->dest);

/* call decision maker */
decmake(15, NULL);
}

void mv89(a)
element *a;
{
    /* 16 */
}

void mv92(a)
element *a;
{
    /* 17 */
    entity *pallet;
pallet = a->pent;

    /* capture the crane itself */
    if (resavail(0) )
        ( recapture(0, 1);

    /* release the conveyor */
    resrelease(6, 1);
    resrelease(7, 1);
    /* schedule a storage */
    schedule(4, now + tasrs, pallet);
}
else
    perror("mv92(): crane was not available");

/* call decision maker */
decmake(17, NULL);
}

void mv95(a)
element *a;


\*
/* 18 */

entity *pallet;
pallet = a->pent;

/* schedule an end of move from 5 to 4 */
if (pallet->dest == 3)
schedule(10, trow + t54, pallet);
else
    perror("mv95(): pallet dest was not 3; it was: ", pallet->dest);

/* call decision maker */
decmaker(18, NULL);
"

void endmc(a)
    element *a;
{
    /* 21 */

    entity *pallet;
pallet = a->pent;

    /* release the machining operation */
    resrelease(4, 1);

    if (pallet->type == 4) /* cnc kit */
    {
        pallet->p[0] = 0;
pallet->p[2] = 0;
pallet->p[3] = 1;
pallet->p[4] = 1;
    }
else if (pallet->type == 5) /* robot kit */
    {
        pallet->p[1] = 0;
pallet->p[2] = 0;
pallet->p[5] = 1;
pallet->p[6] = 2;
    }
else
    perror("endmc(): Errant pallet type");

    /* if the conveyor system is available

203
* then schedule a mv67, reserve the
* conveyor system and release the machining
* system. Otherwise, file the
* pallet in list 3 -- waiting at m/c for conveyor
*/
if (resavell(7))
{
    schedule(11, now + t67, pallet);
    rescapture(7, 1);
}
else
    filefifo(3, pallet);

/* call decision maker */
decmaker(22, NULL);

void endass(a)
element *a;
{
    /* 22 */
    entity *pallet;
pallet = a->pent;

    /* release the assembly operation */
    resrelease(2, 1);

    /* update feeder counters if necessary */
    if ( (pallet->type == 1) /* cnc blanks */
    {
        while ( (feeder < 6) && (pallet->p[0]) )
        {
            feeder++;
            pallet->p[0]--;
        }
    }
    else if ( (pallet->type == 2) /* robot blanks */
    {
        while ( (feederr < 6) && (pallet->p[1]) )
        {
            feederr++;
            pallet->p[1]--;
        }
    }
}
else if ( pallet->type == 3 ) /* link blanks */
{
    while ((feeder1 < 11) && (pallet->p[2]) )
    {
        feeder1++;
        pallet->p[2]--;
    }
}

else if ( pallet->type == 4 ) /* cnc kits */
{
    /* find out what was on pallet before */
    /* assembly station operation and */
    /* update to current contents */
    if ( pallet->p[4] )
    {
        /* pallet had machined parts */
        pallet->p[3] = 0;
        pallet->p[4] = 0;
        pallet->p[7] = 1;
    }
    else
    {
        /* pallet was empty */
        feeder2--;
        feeder1--;
        pallet->p[0]++;
        pallet->p[2]++;
    }
}

else if ( pallet->type == 5 ) /* robot kits */
{
    /* find out what was on pallet before */
    /* assembly station operation and */
    /* update to current contents */
    if ( pallet->p[5] )
    {
        /* pallet had machined parts */
        pallet->p[5] = 0;
        pallet->p[6] = 0;
        pallet->p[8] = 1;
    }
    else
    {
        /* pallet was empty */
        feeder2--;
        feeder1--;
        feeder1--;
        pallet->p[1]++;
    }
}

205
pallet->p[2]++;
pallet->p[2]++;
}
else
    perror("endass(): unknown pallet type: ", pallet->type);

/* set the pallet destiny */
pallet->dest = 0;

/* If the conveyor is available
 * then schedule a mv34, reserve the
 * conveyor system and release the assembly
 * system. Otherwise, file the
 * pallet in list 2.
 */
if (resavail(7))
{
    schedule(5, tnow + t34, pallet);
    recapture(7, 1);
}
else
    filefifo(2, pallet);

/* call decision maker */
decmaker(22, NULL);
}

void decmaker(i, pel)
int i; /* fcn calling decmaker() */
element *pel;
{
    int slf, r3, r5;
    entity *pallet = NULL;

    slf = 1;

    if (resavail(7))
    {
        /* see if any pallet is waiting at assembly or
        * machining for the conveyor
        */
        if ((lb[2].lcur || lb[3].lcur)
        {
            if (lb[2].lcur) /* waiting at assembly */
            {
                pallet = removelst(2);
                schedule(5, tnow + t34, pallet);
            }
        }
    }
else /* waiting at machining */
{
    pallet = remove1st(3);
    schedule(11, tnow + t67, pallet);
}
rescapture(7, 1);
return;
}

pallet = NULL;

if (resavail(7))
{
    /* see if the feeders are low */
    if (feederl < 3 || feeder2 < 2 || feeder3 < 2)
    {
        if (resavail(3) && resavail(0))
        {
            if (feederl < 2 && pins3 == 0)
            {
                if (pallet = findp(3, 2)) == NULL
                    perror("decmaker(): couldn't find the type 3 pallet");
                pins3 = 1;
            }
            else if (feeder2 < 2 && pins1 == 0)
            {
                if (pallet = findp(1, 0)) == NULL
                    perror("decmaker(): couldn't find the type 1 pallet");
                pins1 = 1;
            }
            else if (feeder3 < 2 && pins2 == 0)
            {
                if (pallet = findp(2, 1)) == NULL
                    perror("decmaker(): couldn't find the type 2 pallet");
                pins2 = 1;
            }
        }
        if (pallet)
        {
            pallet->dest = 3; /* assemble op. */
            rescapture(0, 1);
            rescapture(3, 1); /* reserve assemb system */
            rescapture(7, 1); /* reserve conveyor sys */
            schedule(1, tnow + tasrs, pallet);
            return;
        }
    }
    else
    {
        return;
    }
}
pallet = NULL;

if (resavail(7) )
{
    /* check orders and stock */
    if ( (ccur > 0) || (rcur > 0) && resavail(0) )
    {
        r3 = resavail(3);
        r5 = resavail(5);
    
        /* see if more cnc's are needed */
        if ( (ccur - cfin) > (rcur - rfin) )
        {
            if ( (r3>0) || (r5>0) )
            {
                if (r5)
                {
                    if (pallet = findp(4, 0))
                    {
                        /* found pallet with unmachined parts */
                        pallet->dest = 6; /* machining op */
                        rescapture(5, 1); /* reserve m/c system */
                        schedule(1, tnow + tasrs, pallet);
                        rescapture(0, 1); /* capture crane */
                        rescapture(7, 1); /* reserve conveyor sys */
                        return;
                    }
                }
                if (r3)
                {
                    if (pallet = findp(4, 3))
                    {
                        /* find pallet with machined parts */
                        pallet->dest = 3; /* assembly op */
                        rescapture(3, 1); /* reserve assemb system */
                    }
                    else if (pallet = findp(4, -1))
                    {
                        /* found empty pallet */
                        pallet->dest = 3; /* assembly op */
                        rescapture(3, 1); /* reserve assemb system */
                    }
                }
            }
            if (pallet)
            {
                schedule(1, tnow + tasrs, pallet);
                rescapture(0, 1); /* capture crane */
                rescapture(7, 1); /* reserve conveyor sys */
                return;
            }
        }
    }
}
else /* more robots are needed */
{
    if ( (r3>0) || (r5>0) )
    {
        if (r5)
        {
            if (pallet = findp(5, 1))
            {
                /* found pallet with unmachined parts */
                pallet->dest = 6; /* machining op */
                rescapture(5, 1); /* reserve m/c system */
                schedule(1, tnow + tars, pallet);
                rescapture(0, 1); /* capture crane */
                rescapture(7, 1); /* reserve conveyor sys */
                return;
            }
        }
        if (r3)
        {
            if (pallet = findp(5, 5))
            {
                /* find pallet with machined parts */
                pallet->dest = 3; /* assembly op */
                rescapture(3, 1); /* reserve assemb system */
            }
        }
        else if (pallet = findp(5, -1))
        {
            /* found empty pallet */
            pallet->dest = 3; /* assembly op */
            rescapture(3, 1); /* reserve assemb system */
        }
    }
    if (pallet)
    {
        schedule(1, tnow + tars, pallet);
        rescapture(0, 1); /* capture crane */
        rescapture(7, 1); /* reserve conveyor sys */
        return;
    }
}
Appendix C

Outputs of CBST models
Appendix C.1a

Output of single run of M/M/1 queue model

**LIST STATISTICS:** time = 5.00000e+03

<table>
<thead>
<tr>
<th>list</th>
<th>length</th>
<th>waiting time</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>name</td>
<td>ave stdev</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>------</td>
<td>--------------</td>
</tr>
<tr>
<td>event list</td>
<td>2.8950e+00</td>
<td>3.0655e-01</td>
</tr>
<tr>
<td>queue 0</td>
<td>5.3550e+00</td>
<td>5.5190e+00</td>
</tr>
</tbody>
</table>

**RESOURCE STATISTICS:** time = 5.00000e+03

<table>
<thead>
<tr>
<th>resource</th>
<th>utilization</th>
<th>maximum time</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>name cap cur ave stdev</td>
<td>idle busy count</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>server 0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>44.6953</td>
</tr>
</tbody>
</table>
Appendix C.1b

Output of M/M/1 queue model with multiple runs

interarrival time = 10.0, service time = 9.0
expected: u = 0.9000 l = 8.1000 w = 81.0000

Run 0: u = 0.894999 l = 5.3550 w = 50.9373
Run 1: u = 0.872987 l = 4.3730 w = 44.0504
Run 2: u = 0.800966 l = 3.0729 w = 32.6905
Run 3: u = 0.893333 l = 14.1620 w = 132.3586
Run 4: u = 0.841691 l = 3.6436 w = 37.5856
Run 5: u = 0.911661 l = 10.9675 w = 111.2208
Run 6: u = 0.985938 l = 23.6135 w = 240.0729
Run 7: u = 0.862408 l = 7.2111 w = 75.0000
Run 8: u = 0.822366 l = 6.3819 w = 61.2479
Run 9: u = 0.849610 l = 3.5331 w = 36.1194

95 percent confidence intervals

util: 0.8736 +/- 0.0373 [ 0.8363, 0.9109]

length: 8.2314 +/- 4.6232 [ 3.6082, 12.8545]

wait: 92.1274 +/- 46.2987 [ 35.8288, 128.4261]

interarrival time = 10.0, service time = 7.0
expected: u = 0.7000 l = 1.6333 w = 16.3333

Run 0: u = 0.720712 l = 1.8071 w = 18.0302
Run 1: u = 0.775207 l = 1.4772 w = 13.9093
Run 2: u = 0.687031 l = 1.3159 w = 13.3498
Run 3: u = 0.658770 l = 1.4351 w = 13.9878
Run 4: u = 0.660968 l = 1.5131 w = 15.6313
Run 5: u = 0.695673 l = 1.8724 w = 19.1451
Run 6: u = 0.642899 l = 1.4289 w = 15.5650
Run 7: u = 0.707357 l = 1.5088 w = 15.3019
Run 8: u = 0.673227 l = 1.3728 w = 14.7298
Run 9: u = 0.630167 l = 1.0402 w = 10.7430

95 percent confidence intervals

util: 0.6852 +/- 0.0303 [ 0.6549, 0.7155]
interarrival time = 10.0, service time = 5.0
expected: $u = 0.5000$  $l = 0.5000$  $w = 5.0000$

<table>
<thead>
<tr>
<th>Run</th>
<th>u</th>
<th>l</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.359497</td>
<td>0.9530</td>
<td>9.1986</td>
</tr>
<tr>
<td>1</td>
<td>0.556336</td>
<td>0.5923</td>
<td>5.1815</td>
</tr>
<tr>
<td>2</td>
<td>0.444770</td>
<td>0.3377</td>
<td>3.5930</td>
</tr>
<tr>
<td>3</td>
<td>0.467700</td>
<td>0.5250</td>
<td>5.1473</td>
</tr>
<tr>
<td>4</td>
<td>0.430679</td>
<td>0.3635</td>
<td>3.9168</td>
</tr>
<tr>
<td>5</td>
<td>0.440460</td>
<td>0.2900</td>
<td>3.0913</td>
</tr>
<tr>
<td>6</td>
<td>0.456398</td>
<td>0.4825</td>
<td>5.2720</td>
</tr>
<tr>
<td>7</td>
<td>0.517942</td>
<td>0.4122</td>
<td>4.1253</td>
</tr>
<tr>
<td>8</td>
<td>0.481038</td>
<td>0.4448</td>
<td>4.7724</td>
</tr>
<tr>
<td>9</td>
<td>0.492211</td>
<td>0.5932</td>
<td>5.8737</td>
</tr>
</tbody>
</table>

95 percent confidence intervals

util: $0.4847 + - 0.0332$  $[0.4515, 0.5179]$

length: $0.4954 + - 0.1342$  $[0.3613, 0.6296]$

wait: $5.0172 + - 1.2177$  $[3.7995, 6.2349]$

213
Appendix C.2a

CBST output for Cedar Bog Lake model

\[ a \quad 8.3000000e-01 \quad 3.188382e+01 \\
\quad b \quad 3.0000000e+03 \quad 8.438701e+01 \\
\quad c \quad 1.0000000e-04 \quad 7.902532e+01 \\
\quad d \quad 0.0000000e+00 \quad 1.202911e+02 \\
\quad e \quad 0.0000000e+00 \quad 5.441205e+01 \\
0.0000e+00 \quad +abcde \\
2.5000e-02 \quad +edc b a \\
5.0000e-02 \quad +edc b a \\
7.5000e-02 \quad +edc b a \\
1.0000e-01 \quad +edc b a \\
1.2500e-01 \quad +edc b a \\
1.5000e-01 \quad +edc b a \\
1.7500e-01 \quad +edc b a \\
2.0000e-01 \quad +edc b a \\
2.2500e-01 \quad +edc b a \\
2.5000e-01 \quad +edc b a \\
2.7500e-01 \quad +edc b a \\
3.0000e-01 \quad +edc b a \\
3.2500e-01 \quad +edc b a \\
3.5000e-01 \quad +edc b a \\
3.7500e-01 \quad +edc b a \\
4.0000e-01 \quad +edc b a \\
4.2500e-01 \quad +edc b a \\
4.5000e-01 \quad +edc b a \\
4.7500e-01 \quad +edc b a \\
5.0000e-01 \quad +edc b a \\
5.2500e-01 \quad +edc b a \\
5.5000e-01 \quad +edc b a \\
5.7500e-01 \quad +edc b a \\
6.0000e-01 \quad +edc b a \\
6.2500e-01 \quad +edc b a \\
6.5000e-01 \quad +edc b a \\
6.7500e-01 \quad +edc b a \\
7.0000e-01 \quad +edc b a \\
7.2500e-01 \quad +edc b a \\
7.5000e-01 \quad +edc b a \\
7.7500e-01 \quad +edc b a \\
8.0000e-01 \quad +edc b a \\
8.2500e-01 \quad +edc b a \\
8.5000e-01 \quad +edc b a \\
8.7500e-01 \quad +edc b a \\
9.0000e-01 \quad +edc b a \\
9.2500e-01 \quad +edc b a \\
9.5000e-01 \quad +edc b a \\
9.7500e-01 \quad +edc b a \\
\]
Appendix C.2b

SLAM II output for Cedar Bog Lake model

**STATE AND DERIVATIVE VARIABLES**

<table>
<thead>
<tr>
<th>(I)</th>
<th>SS(I)</th>
<th>DD(I)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.2031E+02</td>
<td>0.3523E+02</td>
</tr>
<tr>
<td>2</td>
<td>0.5256E+00</td>
<td>0.3596E+00</td>
</tr>
<tr>
<td>3</td>
<td>0.6811E+00</td>
<td>-0.6192E+00</td>
</tr>
<tr>
<td>4</td>
<td>0.1437E+03</td>
<td>0.5633E+02</td>
</tr>
<tr>
<td>5</td>
<td>0.6498E+02</td>
<td>0.2577E+02</td>
</tr>
<tr>
<td>6</td>
<td>0.1151E+03</td>
<td>0.0000E+00</td>
</tr>
</tbody>
</table>

**TABLE NUMBER 1**

<table>
<thead>
<tr>
<th>TIME</th>
<th>PLANTS</th>
<th>HERBIVOR</th>
<th>Carnivore</th>
<th>Organic</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SOLAR</td>
<td>ES</td>
<td>ES</td>
<td>ES</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ERGY</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minimum</td>
<td>0.0000E+00</td>
<td>0.0000E+00</td>
<td>0.0000E+00</td>
<td>0.0000E+00</td>
<td>0.0000E+00</td>
</tr>
<tr>
<td></td>
<td>0.4200E+02</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum</td>
<td>0.3826E+02</td>
<td>0.1013E+01</td>
<td>0.9481E+00</td>
<td>0.1437E+03</td>
<td>0.6498E+02</td>
</tr>
<tr>
<td></td>
<td>0.1882E+03</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PLOT NUMBER 1**

<table>
<thead>
<tr>
<th>Scales of Plot</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0000E+00</td>
</tr>
<tr>
<td>0.506E+00</td>
</tr>
<tr>
<td>0.718E+02</td>
</tr>
<tr>
<td>0.325E+02</td>
</tr>
<tr>
<td>0.420E+02</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0000E+00</td>
</tr>
<tr>
<td>0.2500E+01</td>
</tr>
<tr>
<td>0.5000E+01</td>
</tr>
<tr>
<td>0.7500E+01</td>
</tr>
</tbody>
</table>
OUTPUT CONSISTS OF 83 POINT SETS (498 POINTS)
STORAGE ALLOCATED FOR 712 POINT SETS (4994 MORDS)
STORAGE NEEDED FOR 83 POINT SETS (581 MORDS)
Appendix C.3

Output of Tank/truck model

time = 5.000, level = 65.000: arrival of truck 1
time = 12.500, level = 80.000: arrival of truck 2
time = 14.000, level = 83.000: arrival of truck 3
time = 15.000, level = 85.000: end of service for truck 1
time = 20.000, level = 95.000: close the bay

LIST STATISTICS: time = 5.30000e+01

<table>
<thead>
<tr>
<th>num</th>
<th>name</th>
<th>ave</th>
<th>stdev</th>
<th>max</th>
<th>cur</th>
<th>ave</th>
<th>stdev</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>q_bay</td>
<td>3.0189e-01</td>
<td>4.9849e-01</td>
<td>2</td>
<td>0</td>
<td>5.3335e+00</td>
<td>7.1824e+00</td>
</tr>
</tbody>
</table>

RESOURCE STATISTICS: time = 5.30000e+01

<table>
<thead>
<tr>
<th>num</th>
<th>name</th>
<th>cap</th>
<th>cur</th>
<th>ave</th>
<th>stdev</th>
<th>idle</th>
<th>busy</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>t_bay</td>
<td>1</td>
<td>5.6604e-01</td>
<td>4.9562e-01</td>
<td>5</td>
<td>20</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

a 6.500000e+01
0.00000e+00 +a
5.00000e-01 +a
1.00000e+00 +a
1.50000e+00 +a
2.00000e+00 +a
2.50000e+00 +a

1.000000e+02
## Index

### A
- `accumarea()`  
  40, 61
- `advance()`  
  43, 50, 51, 64, 65, 66, 72, 73, 75

### B
- `beroulli()`  
  58, 60
- `beta()`  
  58, 59
- `binomial()`  
  58, 60
- `bldarr()`  
  50, 51, 52, 61, 63

### C
- `cancell()`  
  53, 54
- `cancelall()`  
  53, 54
- `checkt()`  
  63, 64
- `clrlist()`  
  61, 62
- `clrov()`  
  61, 62
- `clrrres()`  
  61, 62
- `clrtp()`  
  61, 62
- `combined()`  
  32, 46, 51, 52, 53, 65, 67, 75
- `combpinit()`  
  51, 52
- `continuous()`  
  30, 46, 50, 65, 72
- `contpinit()`  
  43, 50, 52
- `cplot()`  
  61, 63
- `cross()`  
  64

### D
- `desolve()`  
  44, 64, 66
- `discpinit()`  
  36, 46, 52
- `discrete()`  
  27, 46, 47, 53, 71
- `dlast[]`  
  44
- `dnow[]`  
  44, 66

### E
- `entity`  
  26, 42, 53, 54, 55, 56, 68, 69, 71
- `erlang()`  
  58, 59
- `expon()`  
  58, 59

### F
- `filefifo()`  
  55
- `filehvf()`  
  55, 69
- `filelifo()`  
  55
- `filelvf()`  
  55, 69
- `fixevlist()`  
  53
- `fixlist()`  
  55, 56

### G
- `gamlcal()`  
  44, 64, 66
- `gamma()`  
  58, 59
gaussian() 58
getarray() 61, 63

H
hmax 44
hmin 44

I
initlist() 52
insert() 53, 54, 55
insfido() 55
inslifo() 55
insprior() 55, 56

L
listnote 38-39, 52, 55, 56, 62
lognormal() 58, 60

M
main() 43, 46, 47, 50, 51, 70, 78, 79
makeelement() 53, 54
makeentity() 55
mfilestat() 56, 61, 62

N
nextevent() 42, 46, 53
NLISTS 39, 68
NODE 43, 44, 68
normal() 58, 60
notevalue() 41, 61
NR 68
NRES 37, 38, 68
NSEEDS 42, 58, 68
NSTEVP 43, 44, 68

O
obsvar 41, 61

P
perror() 63, 64
perror1() 63, 64
perror2() 63, 64
pfilestat() 55, 56, 61, 62
poisson() 58, 60

R
ran3() 58
removelst() 55, 56
removenum() 55, 56
removespec() 55, 56, 69
replist() 61, 63
<table>
<thead>
<tr>
<th>Function</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>repres()</td>
<td>61, 63</td>
</tr>
<tr>
<td>resavail()</td>
<td>57, 97</td>
</tr>
<tr>
<td>rescapture()</td>
<td>57, 97</td>
</tr>
<tr>
<td>resdecr()</td>
<td>57, 97</td>
</tr>
<tr>
<td>resincr()</td>
<td>57, 97</td>
</tr>
<tr>
<td>resnote</td>
<td>36-38, 57, 59</td>
</tr>
<tr>
<td>resrelease()</td>
<td>57, 97</td>
</tr>
<tr>
<td>rkf()</td>
<td>64, 66</td>
</tr>
<tr>
<td>rhead()</td>
<td>61, 62</td>
</tr>
<tr>
<td>rlist()</td>
<td>61, 62</td>
</tr>
<tr>
<td>rresource()</td>
<td>61, 62</td>
</tr>
<tr>
<td>rrhead()</td>
<td>61, 63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>schedule()</td>
<td>42, 53, 54</td>
</tr>
<tr>
<td>sevents()</td>
<td>72, 74</td>
</tr>
<tr>
<td>sflag[]</td>
<td>44, 74</td>
</tr>
<tr>
<td>slast[]</td>
<td>44, 66</td>
</tr>
<tr>
<td>snow[]</td>
<td>44, 66</td>
</tr>
<tr>
<td>stevcond()</td>
<td>72, 74</td>
</tr>
<tr>
<td>stopcheck()</td>
<td>46, 50, 51, 70, 71</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>tdfull</td>
<td>44</td>
</tr>
<tr>
<td>tdfrec</td>
<td>44</td>
</tr>
<tr>
<td>tevents()</td>
<td>46, 71</td>
</tr>
<tr>
<td>tnow</td>
<td>29, 30, 32, 34, 35, 36, 43, 46, 50, 51, 52, 54, 64, 66</td>
</tr>
<tr>
<td>tol</td>
<td>44</td>
</tr>
<tr>
<td>tpvar</td>
<td>40-41, 61</td>
</tr>
<tr>
<td>triang()</td>
<td>58, 60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>uinit()</td>
<td>36, 37, 43, 44, 46, 50, 51, 56, 70</td>
</tr>
<tr>
<td>uniform()</td>
<td>58, 59</td>
</tr>
<tr>
<td>update()</td>
<td>64, 66</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>whennext()</td>
<td>64, 67</td>
</tr>
<tr>
<td>wiebull()</td>
<td>58, 59</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>yfirst()</td>
<td>72, 81</td>
</tr>
</tbody>
</table>
Vita

Fazal Ullah Khan was born on March 16, 1963 in Karachi, Pakistan. The author reached the United States in 1967 and has lived here since then. He graduated from George Washington Senior High School in Charleston, West Virginia in June, 1981. He graduated with a B. S. in Industrial Engineering from West Virginia University in May, 1987. He plans to obtain an M. S. in Industrial and Systems Engineering in February, 1991. Afterwords, he plans on working in industry and becoming a significant contributor to the field of simulation.

Fazal U. Khan