User's Guide

CSIM for Java Simulation Engine

April 2006

Mesquite Software, Inc.
P.O. Box 26306
Austin, TX 78755-0306
+ 1 (512) 338-9153
Internet: info@mesquite.com

Table of Contents

Table of Contents

Preface

Welcome to *CSIM for Java*, based on Mesquite Software's CSIM 19 simulation software for the C and C++ programming languages. CSIM has been helping C and C++ programmers build fast, efficient *discrete-event* simulations for more than 20 years, and is now available for Java programmers.

Used by thousands of customers worldwide, CSIM is a fully functional, proven software toolkit for programmers who need to simulate the performance of *process-oriented, event-based* systems. CSIM's powerful simulation capabilities benefit programmers, groups, departments, managers, and companies by enabling the evaluation of large, complex systems, thus improving system design and operation.

CSIM provides several key benefits over other simulation solutions:

- *Flexible standard programming environment* – very short learning curve for programmers, with no proprietary environment to learn
- *Stable, proven software* – much faster and less expensive than building one's own simulator "from scratch"
- *Fast execution, low overhead*
- *Industrial strength*, no inherent limitations – ideal for large applications (300,000 events plus!)
- *Integrates easily* with existing code
- *Excellent value* – $1195 per seat or less

CSIM was originally developed in 1985 at the Microelectronics and Computer Technology Corporation (MCC) in Austin, Texas. Dr.

Herb Schwetman helped develop the original version and acquired the rights to CSIM when he founded Mesquite Software in 1994. Since then, thousands of users at major international corporations and universities have successfully used CSIM to study and develop complex systems, such as:

- Network Protocols and Systems
- Telecom Communication Systems
- Aerospace and Defense Systems
- Software and Hardware Applications
- Manufacturing and Transportation Processes
- And much more...

This *User's Guide* describes how to use CSIM for Java. You can find the latest and greatest version of this guide, along with additional helpful information, on the Mesquite Software website at http://www.mesquite.com/documentation/index.htm.

CSIM for Java Functionality

CSIM for Java is a library of classes, functions, procedures and header files that enable developers to implement efficient models of complex systems. The models are written in the Java programming language. Typically, such a model is used to provide estimates of the performance of the real (modeled) system. Many times, the goal is to use the model to "try out" different system configurations or different workloads or different resource scheduling rules, in order to find the "best" configuration (or workload or schedule) with respect to the performance goal.

Many simulation applications involve entities passing through a system of queues. In CSIM, entities are represented by *processes*, and queues are represented by *facilities* and other simulated resources. In these models, the complete system is represented by a collection of simulated resources and a collection of processes that compete for use of these resources.

CSIM for Java provides the complete infrastructure required to run discrete-event simulations, including many additional features that enhance control and save time:

- *Facility* structures to model resources (servers) and waiting processes
- Efficient mechanism for creating and destroying *processes*
- Processes have *private data store (memory*) and *access to global data*
- User has the ability to *control the simulated time* in which the code executes
- *Built-in threads package* and *scheduler*
- Programmer can *specify service discipline* (such as *FCFS* – first come, first served) and can *set process priority* at a facility to control the way processes are scheduled at that facility
- *Storages*, *buffers*, *events* and *mailboxes* that support process activities
- *Automatic reports* are generated for facilities on many parameters, such as service time, throughput rate, average queue length, and average response time. They are also generated for storages, buffers, events, and mailboxes. Additional data-gathering can be easily configured.
- *Tables* and *qtables* collect data during model execution
- Ability to calculate *confidence intervals* using the method of batch means
- *Run length control* provide a mechanism for running a model until a desired confidence level has been achieved for a specified statistic

- Ability to *trace model execution*
- Many *statistical distribution functions* that simulate "real-life" randomness, such as customer arrival rates, part failure rates, etc.

CSIM for Java provides these structures so that users can spend their time designing the model, writing process descriptions and focusing on the sequence of events. CSIM for Java's use of a standard programming language makes it possible to embed models in other products for flexibility in output, reporting, and user interaction with processes. For example, models have been embedded in systems to present information graphically and used to train people in the operation of the system.

Unlike other simulation software with intricate user interfaces and proprietary environments, CSIM for Java's standard programming environment poses only a slight learning curve to programmers, integrates easily with existing programs, and executes quickly with low overhead.

Typographical Conventions

To help distinguish between function names, example code, CSIM output, and other text, this guide follows the typographical conventions described in the following table.

Convention	Meaning
bold	**Example** and **Prototype** headers, **icon**, **tab**, and **button** names
italic	*Function names*, *reserved words*, *key words* being introduced for the first time,

	chapter and *section names*, *Notes* and *Cautions*, and any other text that needs particular emphasis.
`Courier font`	`Code examples` and `filenames`.
Arial 8 point font	Output from a CSIM model

System Requirements and Supported Platforms

CSIM for Java runs any system with Java and a JVM installed. Java development kit (jdk) version 1.4.1 or higher is required.

Installation

CSIM for Java is distributed as a Java Archive (jar) file. There are at least three different ways of installing and using CSIM for Java: using the Command line, using the Eclipse interactive development environment (IDE), and with the NetBeans IDE.

You may need to install a Java SDK (jdk) or Java Runtime Environment (JRE) on your system. One such jdk is available from Sun Microsystems, Inc., at www.sun.com –>**downloads** –> **Java 2 Standard Edition** –> **J2SE 5.0**.

Command Line

1) Create a directory called `csimForJava`.

2) Copy the `csimForJava.jar` file to this folder.

3) Copy the Java program file, `App.java`, to this folder.

4) Using the Command Prompt window, execute the following statements:
   a. `javac –classpath csimForJava.jar;. App.java`
   b. `java –classpath csimForJava;. App`

   Note: Make sure to include the ";." in this step; without this syntax, the statements will not execute.

5) The output for `App.java` is in the file `App.out`

On Windows, you may have to alter the path environment variable, and append the directory that holds the `javac` and `java` executables; the command for doing this is:

```
set path=%path%;c:\<insert full path here>
```

So, for example:

```
set path=%path%;c:\j2sdk1.4.2_01\bin
```

Note: This changes the path variable in the current window.

You can also modify the path variable for every Command Prompt window in the Windows Control Panel:

1) Open the **System** control panel icon

2) Select the **Advanced** tab

3) Click the **Environment Variables** button.

Note: Using the Cygwin has not been verified as a good technique for running Java applications and the `csimForJava.jar`.

Eclipse

1) In Eclipse, create a new project called `CsimForJava`.
2) Import a java file (e.g., `App.java`).
3) Add the `csimForJava.jar` file:
   a. Right click on the project (in the Package Explorer)
   b. Click **Properties**
   c. In the left pane, click Java Build Path
   d. Select the **Libraries** tab
   e. Select **Add External JARs**, find the `csimForJava.jar` file, and add it.
4) To run the program, right click the source file. Using the **Run** option, select **Java Application**.
5) The output for the `App.java` program is in the file `App.out` `App.txt`.

NetBeans

1) In NetBeans, create a new project. For these instructions, assume that the name of this project will be `CsimForJava`.
   a. Select **General project** and then **Java Application**
   b. Disable the **Create Main Class** option
2) In the new directory named `CsimForJava`
   a. Copy the file `App.java` to the `src` directory in `CsimForJava`
   b. Copy the `csimForJava.jar` file to `CsimForJava`
3) In the NetBeans project called `CsimForJava:`

a. In the folder name `Source Packages`, you should find the file `App.java` under the `<default>` package
b. Right click the `Libraries` folder; select the **Add Jar/folder...** option and add the `csimForJava.jar` file to the project

4) Right click the `<default>` folder and select **Compile Package**

5) Right click the `CsimForJava` folder and select **Run Project**; click **OK** when asked to choose the `Main` class
6) When execution ends, use the **File -> Open** menu to open the `App.out` file in order to view the results.

Support and Contact Information

If you have questions about CSIM or would like to purchase a copy, please contact Mesquite Software or visit our website at www.mesquite.com.

Mesquite Software
8500 N. Mopac Expwy, Suite 825
Austin, Texas 78759 USA

Phone: (800) 538-9153 in US or +1 (512) 338-9153
Fax: +1 (512) 338-4966
E-mail: info@mesquite.com
Web: www.mesquite.com

1. Introduction

CSIM for Java is a process-oriented discrete-event simulation package for use with the Java programming language. CSIM for Java is based on CSIM[1], a simulation library for use with C/ C++ programs. CSIM for Java is implemented as a library of classes that implement a comprehensive set of structures and operations. The end result is a convenient tool that programmers can use to create simulation models of complex systems.

In this User's Guide, the term "CSIM" refers to CSIM for Java; if CSIM for C/C++ is referenced, it will be specified as such.

A CSIM program models a system as a collection of CSIM processes that interact with each other by using CSIM objects. The purpose of modeling a system is to produce estimates of time and performance. The model maintains simulated time, so that the model can yield insight into the dynamic behavior of the modeled system.

This document provides a description of:

- CSIM objects and the statements that manipulate them
- Reports available from CSIM
- Information on compiling, executing and debugging CSIM programs.

[1] CSIM is copyrighted by Microelectronic and Computer Technology Corporation, 1985-1994. Java is a registered trademark of Sun Microsystems, Inc.

1.1. Java Implementation of CSIM

The goal for the Java implementation of CSIM is to give Java programmers a set of classes and objects that can be used to implement process-oriented, discrete-event simulation models. CSIM is used as the basis for this framework. CSIM C/C++ has been shown to be a flexible, powerful and useful library for use by C++ and/or C programmers to develop simulation models of large, complex systems. Using CSIM C/C++ as a basis for CSIM for Java extends these capabilities to the Java environment.

CSIM for Java is a Java toolkit. It makes use of Java threads to implement CSIM processes. Java programmers should find this toolkit to be a natural and convenient way of implementing simulation models. CSIM C/C++ programmers should find many familiar concepts and mechanisms in the Java implementation.

1.2. CSIM Objects and the Model Object

CSIM provides a number of classes that define the objects used by CSIM models. A CSIM model is a class that extends the Model class.

At a high level, CSIM provides the following simulation objects:

- *Processes* – the active entities that request service at facilities, wait for events, etc. (*i.e.*, processes deal with all of the other structures in this list)

- The *Facility* class – queues and servers reserved or used by processes
- The *Storage* class – resources that can be partially allocated to processes
- The *Buffer* class – resources that can be partially allocated to processes; a buffer can be empty and/or partially full
- The *Event* class – used to synchronize process activities
- The *Mailbox* class – used for inter-process communications
- Data collection structures – used to collect data during the execution of a model; these include the *Table* class and the *QTable* class
- The *ProcessClass* class – used to segregate statistics for reporting purposes
- The *Random* class – implements streams of random numbers

The processes mimic the behavior of active entities in the simulated system.

The *Model* class has a number of methods that are used as a CSIM model is created. These methods are noted in the following sections. The *Model* class is described in Chapter 19 – *Output From CSIM and the Model Class*.

1.3. Syntax Notes

- All parameters are required.
- Whenever a parameter is included within double quotes (*e.g.*, "name"), it can also be passed as a string.

1.4. Import Files

All of the classes required by a CSIM model are defined in a set of import files. The usual set of import files is as follows:

```
import com.mesquite.csim.*;
import com.mesquite.csim.Process;
import com.mesquite.csim.file.Files;
import java.io.*;
```

The class Process is in conflict with class Process in java.lang.Process; thus the class Process in CSIM for Java must be explicitly imported.

The class Random in java.util.Random is in conflict with the class Random in CSIM for Java. The CSIM version overrides; however, if java.util.Random is used, it must referenced using the full path name.

1. Introduction

2. Simulation Time

Time is an important concept in any performance model. CSIM maintains a simulation clock whose value is the current time in the model. This simulation time is distinctly different than the CPU time used in executing the model or the "real world" time of the person running the model. Simulation time starts at zero and then advances unevenly, jumping between the times at which the state of the model changes. It is impossible to make time move backwards during a simulation run.

The simulation clock is implemented as a double precision floating point variable in CSIM. For most models, there is no need to worry that the simulation clock will overflow or that round-off error will impact the accuracy of the clock.

The simulation clock is used extensively within CSIM to schedule events and to update performance statistics. CSIM processes may retrieve the current time for their own purposes and may indirectly cause time to advance by performing certain operations.

2.1. Choosing a Time Unit

The CSIM simulation clock has no predefined unit of time. It is the responsibility of the modeler to choose an appropriate time unit and to consistently specify all amounts of time in that unit. All performance statistics reported by CSIM should also be interpreted as being in that chosen time unit.

A good time unit might be close to the granularity of the smallest time periods in the model. For example, if the smallest time periods being modeled are on the order of tens of milliseconds, an appropriate time unit might be either milliseconds or seconds. Using microseconds or minutes as the time unit would produce performance statistics that are either very large or very small numbers.

Most numbers appearing in CSIM performance reports are printed with up to six digits to the left of the decimal point and six digits to the right of the decimal point. A time unit should be chosen to avoid numbers so large that they overflow their fields or so small that interesting digits are not visible.

2.2. Retrieving the Current Time

The *clock* method from the *Model* class can be used to retrieve the current value of the simulation clock, as follows:

Prototype:
```
public class MyModel extends Model {
  public static void main(String args[]) {
    MyModel model = new MyModel();
```

Example:
```
double x = model.clock();
```

NOTE: Because a process is always contained in a class derived from the *Model* class, the following is allowed:

Example:
```
double x = clock();
```

2.3. Delaying for an Amount of Time

A CSIM process can delay for a specified amount of simulation time by calling the *hold* function.

Prototype: `void hold(double amount_of_time)`

Example: `hold(1.0);`

The *hold()* method is part of the *Process* class. As references to the *hold()* method are normally made from within a *Process* object, the reference does not have to qualified by the containing class.

If there are other processes waiting to run, the calling process will be suspended. Otherwise, time will immediately advance by the specified amount.

A process can delay until a specified time by calling *hold* with a parameter value equal to the specified time minus the current time. To make a simulation begin with a clock value other than zero, simply call *hold* at the beginning of the *sim* function with an amount of time equal to the desired initial time.

Calling the *hold* function with a zero amount of time might at first seem to be meaningless. But, it causes the running process to relinquish control to any other processes that are waiting to run at the same simulation time. This behavior can be used to affect the order of execution of processes that have activities scheduled for the same simulation time.

2.4. Advancing Time

There is no way for a program to directly assign a value to the simulation clock. The simulation clock advances as a side effect of a process performing one of the following method-calls:

`process.hold`	`storage.allocate`
`buffer.get`	`buffer.put`
`buffer.timed_put`	`buffer.timed_get`
`event.untimed_wait`	`event.queue`
`event.wait_any`	`event.queue_any`
`event.timed_wait`	`event.timed_queue`
`facility.reserve`	`facility.timed_reserve`
`facility.use`	`storage.timed_allocate`
`mailbox.receive`	`mailbox.timed_receive`

Calling one of these methods does not guarantee that time will advance. For example, calling the *storage.allocate* method will cause time to pass only if the requested amount of storage is not available.

All CSIM method calls not listed above, as well as all Java language statements, occur instantaneously with respect to simulation time. A CSIM program can perform arbitrarily many activities in a single instant of simulation time.

A common programming error is to create a CSIM process that calls none of the methods in the above list. When this process receives control, it runs endlessly to the exclusion of all other CSIM processes.

2.5. Displaying the Time

There are several ways the simulation time can be automatically displayed while running a CSIM program. Every trace message contains the current simulation time. The *model.clock()* method can be used to get the current simulated time. Also, when the *report* function is called to produce a report of all statistics, the report header contains the current simulation time.

2.6. Integer-Valued Simulation Time

In some simulation models, *e.g.*, models of computer hardware, it is the case that time can only assume discrete integer values. Although CSIM maintains time as a floating-point variable, some simple programming techniques can insure that the clock will always have an integer value. (Here, we are using the word *integer* in the mathematical sense.) Amounts of time appear as input parameters in calls to the following methods: *process.hold*, *facility.use, facility.timed_reserve, storage.timed_allocate, buffer.timed_put, buffer.timed_get, mailbox.timed_receive, event.timed_wait*, *event.timed_queue, event.time_wait_any* and *event.timed_queue_any*. To maintain an integer-valued clock, these parameters must have values that are integers (although of type *double*), which can be accomplished either

by specifying an integer numeric literal or by using an integer-valued function.

Example: `hold(10);`

Example: `bus.use(uniform_int(1,5));`

Example: `bus.use(Math.floor(exponential(1.0)));`

The automatic type conversion features of Java insure the correct results.

The IEEE Floating Point Standard guarantees that addition and subtraction with integer-valued operands will yield integer-valued results. CSIM performs only addition on the simulation clock.

3. Processes

Processes represent the active entities in a CSIM model. For example, in a model of a bank, customers might be modeled as processes (and tellers as facilities). In CSIM, a process is an extension of the *Process* class; the *Process* class extends the *java.lang.Object* class. A CSIM process should not be confused with a UNIX process (which is an entirely different thing). The first process in a CSIM model is invoked using the *model.start* method; all subsequently activated processes are invoked using the *model.add* method. A process can be invoked with input arguments, but it cannot return a value to the invoking process.

There can be several simultaneously "active" instances of the same process. Each of these instances appears to be executing in parallel (in simulated time) even though they are in fact executing sequentially on a single processor. The CSIM runtime package guarantees that each instance of every process has its own runtime environment. All processes have access to a program's global variables.

A CSIM process, just like a real process, can be in one of four states:

- Actively computing
- Ready to begin computing
- Holding (allowing simulated time to pass)
- Waiting for an event to happen (or a facility to become available, etc.)

When an instance of a process terminates, either explicitly or via a method exit, it is deleted from the CSIM system. Each process has a unique process id and each has a priority associated with it.

3.1. Initiating a Process

In CSIM, a *process* is a class that extends the *Process* class. The first process in the model is initiated using the *model.start(Process p)* method:

Prototype: `private class Proc extend Process { };`

Example: `model.start(new Proc);`

Subsequent processes are initiated (or invoked, or started) when another process calls the *model.add(Process p)* method:

Prototype: `private class Proc extend Process { };`

Example: `model.add(new Proc);`

Caution: A process cannot return a function value.

3.2. The *model.add* and *model.start* Methods

The *model.start(new Proc)* and the *model.add(new Proc)* methods (see above) each instantiate an instance of the process *Proc*. When the *start* or *add* method is called, the following actions take place:

- A process control block for the new process (*Proc*) is created and scheduled for execution at the current point in simulated time, and
- The invoking process (the process calling the *add* method) continues its execution (*i.e.*, it remains the actively computing process) at the statement after the call to the *add* method.

The calling process continues as the active process until it suspends itself.

No simulated time passes during the execution of an *add* (or *start)* method call.

Every process instance in a CSIM model is assigned an almost-unique process id. Process id's are 32 bit integers; if 2^{31}-1 id's are used, the sequence of id's is reset to 2.

In many cases, the first process started is a process named *sim*, but this is not required.

3.3. The Structure of a Process

Every CSIM process must have a *constructor* method and must implement a *run* method. The new process extends the *Process* class provided by CSIM. The constructor calls the *Process* class constructor using the *super(String name)* method. The *run()* method contains the statements that implement the "behavior" of the new process.

Example:
```
private class Job extends Process {
   public Job() {
      super("Job");
   }
   public void run() {
      . . . .  // statements
   }
}
```

If the calling process needs to transmit input arguments (parameters) to the new process, it would use the constructor and would store the arguments as local variables internal to the new process:

Example:
```
private class Job extends Process {
   public Job(int arg1, double arg2) {
      super("Job");
      m_arg1 = arg1;
      m_arg2 = arg2:
   }
   int m_arg1;
   double m_arg2;
   public void run() {
      if(m_arg1 ==  0) {
      ...
      }
   }
}
```

3.4. Process Operation

Processes appear to operate simultaneously with other active processes at the same points in simulated time. The CSIM process manager creates this illusion by starting and suspending processes as time advances and as events occur. Processes execute until they "suspend" themselves by doing one of the following actions:

- Execute a *hold* statement (delay for a specified interval of time),
- Execute a statement that causes the processes to be placed in a queue, or
- Exit (the process).

Processes are restarted when the time specified in a *hold* statement elapses or when a delay in a queue ends. It should be noted that

simulated time passes only by the execution of *hold* statements. While a process is actively computing, no simulated time passes.

The Java process object manager preserves the correct context for each instance of every process. In particular, separate versions of all local variables and input arguments for each process are maintained.

3.5. Terminating a Process

A process terminates when the *run* method does a normal method exit or when a process executes the *Process.terminate()* method.

Prototype: `void Process.terminate()`

Example: `terminate();`

The normal case is for a process to do a normal method exit. The *terminate* method is provided when this normal case is not appropriate.

3.6. Changing the Process Priority

The initial priority of a process is inherited from the initiator of that process. For the first process, the default priority is 1 (low priority).

Prototype: `void set_priority(long new_priority)`

Example: `set_priority(5);`

This statement can appear anywhere in the *run()* method for a process. Lower values represent lower priorities (*i.e.*, priority 1

processes will run after priority 2 processes when priority is a consideration in order of execution. Process priorities are used to order processes waiting in queues (*e.g.*, queues in facilities or queues in events). In case of equal priorities, a process joining a queue is placed after the other processes with equal priorities. If all process priorities are equal, the waiting processes are serviced in first come, first served (FCFS) order.

3.7. Inspector Functions

These functions each return some information to the process that issues the statement. The type of the returned value for each of these functions is indicated.

Prototype:	**Functional Value:**
`String name()`	retrieves pointer to name of process issuing inquiry
`int identity()`	retrieves the identifier (process number) of process issuing the inquiry
`int priority()`	retrieves the priority of the process issuing inquiry

4. Facilities

A *facility* is normally used to model a *resource* (something a process requests service from) in a simulated system. For example, in a model of a computer system, a CPU and a disk drive might both be modeled by CSIM facilities. A simple facility consists of a single server and a single queue (for processes waiting to gain access to the server). Only one process at a time can be using a server. A *multi-server* server facility contains a single queue and multiple servers. All of the waiting processes are placed in the queue until one of the servers becomes available.

Normally, processes are ordered in a facility queue by their priority (a higher priority process is ahead of a lower priority process). In cases of ties in priorities, the order is first come, first served (FCFS). Service disciplines other than priority order can be established for a server. These are described in section 4.10, *Specifying the Service Discipline at a Facility*.

Briefly, every facility is declared with a service type. Normally, a facility is declared as a facility of specified type (*e.g.*, *FCFSFacility*). However, a basic facility (*Facility*) can be instantiated as a typed facility; this is useful in dealing with an array of facilities of different types.

A set of usage and queueing statistics is automatically maintained for each facility in a model. The statistics for all facilities that have been used are "printed" when either a *report* (see section 19.3, *CSIM Report Output*) or a *report_facilities* is executed (see section 4.4, *Producing Reports*, for details about the reports that are generated). In addition, there is a set of inspector methods that can be used to extract individual statistics for each facility.

First time users of facilities should focus on the following four sections, which explain how to set up facilities, use (and reserve and release)

facilities, and produce reports. Subsequent sections describe the more advanced features of facilities.

4.1. Declaring and Initializing a Facility

A single-server, first come, first served (FCFS) facility is declared as follows:

Example: `FCFSFacility m_fac("facName");`

A newly created single-server facility is created with a single server that is "free". The facility name is used only to identify the facility in output reports and trace messages. Facilities using other types of scheduling disciplines (other than FCFS) are available (see below).

Facilities are normally declared as globally accessible (accessible to all of the processes in the model).

4.2. Using a Facility

A process typically uses a server for a specified interval of time.

Prototype: `void use(double);`

Example: `m_fac.use(exponential(1.0));`

If the server at this facility is free (not being used by another process), then the process gains exclusive use of the server and the usage interval starts immediately. At the end of the usage interval, the process gives up use of

the server and departs this facility. Execution continues at the statement following the *use* statement.

If the server at this facility is busy (is being used by another process), then the newly arriving process is placed in a queue of waiting processes; this queue is ordered by process priority, with processes of equal priority being ordered by time of arrival. As each process completes its usage interval, the process at the head of the queue is assigned to the server and its usage interval starts at that time.

The service discipline at a facility specifies how processes are given access to the server. One of several different service disciplines can be specified for a facility. And, another form of facility, the *multi-server* facility, has multiple servers. In addition, it is possible to have an *array of facilities*. The difference between a multi-server facility and an array of facilities is that a multi-server facility has one queue for all of the waiting processes, while an array of facilities has a separate queue for each facility in the array.

4.3. Reserving and Releasing a Facility

In some cases, a process will acquire a server, but will do something other than enter the usage interval when it gets the server. The statements for doing this are *reserve* (to gain exclusive use of a server) and *release* (to relinquish use of the server acquired in a previous *reserve* statement).

Prototypes:
```
int reserve();
void release();
```

Examples:
```
m_fac.reserve();
m_fac.release();
```

When a process executes a *reserve*, it either gets use of the server immediately (if the server is not busy) or it is suspended and placed in a

queue of processes waiting to get use of the server. When it gains access to the server, it executes the statement following the *reserve* statement. Processes are ordered in the queue by priority, with processes of equal priority being ordered by time of arrival. This process priority service discipline is called *FCFS* in CSIM; it is the only service discipline that can be specified for facilities where processes do this *reserve-release* style of access. If another service discipline is in force, then the processes must execute *use* statements instead of *reserve-release* pairs of statements.

The value returned is the index of the server assigned to the process. Normally, this is not needed and can be ignored.

The process that releases a server at a facility must be the same process as the one that reserved it. When a process executes a *release*, it gives up use of the server. If there is at least one process waiting to start using the server (*i.e.*, there is at least one process in the queue at this facility), the process at the head of the queue is given access to the server and that process is then reactivated and will proceed by executing the statement following its *reserve* statement. No simulation time passes during execution of a *release* statement.

Note: Executing the sequence *m_fac.reserve(); hold(t); m_fac.release();* is equivalent to executing the statement *m_fac.use(t).* However, if the usage interval is specified by a random number function, then there is a subtle difference between these functions: the randomly derived interval is determined *after* gaining access to the server in the first sequence and *before* gaining access to the server with the *use* form. Thus, it is likely that the intervals in these two examples will be different. In other words, the sequence *m_fac.reserve(); hold(exponential(t); m_fac.release();* will not necessarily exhibit exactly the same behavior as executing the statement *m_fac.use(exponential(t));*.

4.4. Producing Reports

Reports for facilities are most often produced by calling the *report* function, which prints reports of all the CSIM objects. Reports can be produced for all existing facilities by calling the *report_facilities* method in the *model* class.

Prototype: `void report_facilities();`

Example: `model.report_facilities();`

The report for the set of facilities, as illustrated below, includes for each facility: the name of the facility, the service discipline, the average service time, the utilization, the throughput rate, the average queue length, the average response time and the number of completed service requests.

It should be noted that the queueing statistics for a facility uses the convention from queueing theory in which the customer(s) in service are counted as part of the average queue length and the service time for customers are included in the average response time. The average time in the queue but not in service can computed by subtracting the average service time from the average response time. Similarly, the average number of customesr in the queue but not in service can be computed by subtracting the utilization from the average queue length.

FACILITY SUMMARY

facility name	service disc	service time	util.	through put	queue length	response time	compl count
f	fcfs	0.40907	0.208	0.50900	0.27059	0.53162	509
ms fac	fcfs	1.50020	0.764	0.50900	0.83821	1.64678	509
> server 0				1.55358	0.494		0.31800 318
> server 1				1.41133	0.270		0.19100 191
q	rnd_rob	0.73437	0.507	0.69000	0.95522	1.38438	690

4.5. Resetting a Facility

In some cases, it is necessary to reset the statistics counters for a specific facility.

Prototype: `void reset()`

Example: `m_fac.reset();`

Executing this statement does not affect the state of the facility or its servers. The *reset()* and the *reset_facilities* statements each call *facility::reset()* for all facilities in the model.

4.6. Declaring and Initializing a Multi-server Facility

In some cases, a facility has multiple servers, and each of these servers is indistinguishable from the other servers. A mutli-server facility is declared:

A multi-server facility is constructed as follows:

Prototype: `FCFSFacility_ms(char *name, long ns);`

Static Example: `FCFSFacility cpus("dual cpu", 2);`

A process can either execute a *use* statement or the *reserve-release* pair of statements at a multi-server facility. In either case, the process gains access to any server that is free. A process is suspended and put in the single queue at the facility only when all of the servers are busy.

4.7. Releasing a Specific Server at a Facility

Sometimes, it is necessary for one process to reserve a facility and then for another process to release the server obtained by the first process. In this case, the first process has to save the index of the server it obtained, and then give this server index to the second process, so that it can specify that index in the *release(index)* statement, as follows:

Example: `server_index = m_fac.reserve();`

Prototype: `void release(int serverIndex)`

Example: `m_fac.release(serverIndex);`

This command operates in the same way as the *release* statement, except that the ownership of the server is not checked; thus, a process that did not reserve the facility may release it by executing the *release(index)* statement with a server index.

4.8. An Array of Facilities

An array of facilities can be initialized as follows:

Example:

```
Facility m_fac[];
m_fac = new Facility[5];
for(i = 0; i < 5; i++)
    m_fac[i] = new FCFSFacility("fac"+i);
```

In an array of facilities, each element is an independent, single server facility, with its own queue. Each of these facilities is given a constructed name that shows its position in the set. In the above example, the name for the first element of the set is *fac0*. Arrays of facilities are used to model cases where each server has its own queue of waiting processes.

An individual element of a facility is accessed as an array element, as follows:

Example:

```
m_fac[i].use(exponential(1.0));
```

4.9. Reserving a Facility with a Time-out

Sometimes a process must not wait indefinitely to gain access to a server. If a process executes the *timed_reserve* method, it will be suspended until either it gains use of a server or the specified time-out interval expires.

Prototype: `long timed_reserve(double timeout)`

Example:
```
result = m_fac.timed_reserve(100.0);
if (result == -1) . . .
```

The process must check the functional value to determine whether it obtained a server. If the returned value is -1, the process did not obtain a server. If the returned value is not -1, then the process did obtain a server and should eventually release the server.

4.10. Specifying the Service Discipline at a Facility

The service discipline for a facility determines the order in which processes at the facility are given access to that facility. The most commonly used service discipline for a facility is *FCFS*. When the priorities differ, processes gain access to the server in priority order (higher priority processes before lower priority). When processes have the same priority, the processes gain access in the order of their arrival at the facility (first come, first served). Facilities with different service disciplines are different types of facilities. The types of facilities are as follows:

- First come, first served *FCFSFacility*

- Infinite capacity server *INFFacility*
- Last come, first served *LCFSFacility*
- Pre-empt resume *PRERESFacility*
- Round robin pre-empt *RNDPREFacility*
- Round robin priority *RNDPRIFacility*
- Round robin *RNDROBFacility*

Prototypes:
```
FCFSFacility (“name”)
INFFacility (“name”);
LCFSFacility (“name”);
PRERESFacility (“name”);
RNDPREFacility (“name”);
RDNPRIFacility (“name”);
RNDROBFacility (“name”);
```

Example:
```
RNDROBFacility m_rndRobFac(“cpu”);
```

This service function can be any of the following pre-defined service discipline functions:

- *FCFSFacility* – first come, first served

  This is the normal service discipline and is described in the introduction to this section.

- *INFFacility* – infinite servers

  There is no queueing delay at all since there is always a server available at the facility.

- *LCFSFacility* – last come, first served, pre-empt

  Arriving processes are always serviced immediately, pre-empting a process that is currently being served if necessary. Priority is not a consideration with this service discipline.

- *PRERESFacility* – pre-empt resume

  Higher priority processes will pre-empt lower priority processes, so that the highest priority process at the facility will always finish using it first. Where the priorities are the same, processes will be served on a first come, first served basis. Pre-empted processes will eventually resume and complete their service time interval.

- *RNDPREFacility* – round robin with pre-emption

- *RNDPRIFacility* – round robin with priority

  Higher priority processes will be served first. When there are multiple processes with the same priority, they will be serviced on a round robin basis, with each getting the amount of time specified in *set_timeslice* (see below) before being pre-empted by the next process of the same priority.

- *RNDROBFacility* – round robin

  Processes will be serviced on a round robin basis, with each getting the amount of time specified in *set_timeslice* (see below) before being pre-empted by the next process requiring service. The default value of the time slice interval is 1.0. Process priority is not a consideration with this service discipline.

Caution: The *use* statement (as opposed to the *reserve*) statement must be used for most of these service disciplines to be effective. Only *FCFS* will operate properly with *reserve.*

To set the time slice for the round robin service disciplines, *RNDPREFacility, RNDPRIFacility* and *RNDROBFacility* (see above):

Prototype: `void timeslice(double slice_length)`

Example: `m_fac.set_timeslice(0.01);`

4.11. Collecting Class-Related Statistics

Information about usage of a facility by processes that belong to different process classes can be collected for all facilities or for a specific facility (see Chapter 17 for information on process classes). To collect class-based usage information for a specific facility:

Prototype: `void collect()`

Example: `m_fac.collect();`

Usage of this facility by all process classes will be reported in the *facilities* report. Also, it is an error to change the maximum number of classes allowed after this statement has been executed.

To collect usage information for all facilities:

Prototype: `void collect_class_facility_all()`

Example: `model.collect_class_facility_all();`

This command applies to all of the facilities in existence when this statement is executed. Usage of the facilities by all process classes will be reported in the *facilities* report. It is an error to change the maximum number of classes allowed after this statement has been executed.

4.12. Inspector Methods

All statistics and information maintained by a facility can be retrieved during execution of a model or upon its completion.

Prototype:	**Functional Value:**
`String name()`	name of facility

`long numServers()`	number of servers at facility
`String type()`	name of service discipline at facility
`double timeslice()`	time in each time-slice for facility (which has a round robin service discipline)
`int num_busy()`	number of servers currently busy at facility
`int qlength()`	number of processes currently waiting at facility
`int completions()`	number of completions at facility
`double queueLength()`	mean queue length at facility
`double responseTime()`	mean response time at facility
`double serviceTime()`	mean service time at facility
`double throughput()`	mean throughput rate at facility
`double utilization()`	utilization (fraction of time busy) at facility

Data on a facility is available in a *FacilityStats* object; a *FacilityStats* object can be obtained as follows:

```
FacilityStats stats = m_fac.stats(n)
```

facility statistics object

The elements in a *FacilityStats* object are:

`int stats.completions()`
number of completions at facility

`double stats.serviceTime()`
mean service time at facility

`double stats.throughput()`
mean throughput rate at facility

`double stats.utilization(long sn)`
utilization at facility

Additional data on servers can be obtained as follows:

`ServerStats stats = m_fac.serverStats(n)`
statistics object for server *n* at facility

`int stats.completions()`
number of completions for server *n* at facility

`double stats.serviceTime()`
mean service time for server *n* at facility

`double stats.throughput()`
mean throughput rate for server *n* at facility

`double stats.utilization(long sn)`
utilization for server *n* at facility

Data on the utilization of a server by a specific process class can be obtained as follows:

`ProcessClass cl`
process class object

`ServerStats stats = m_fac.stats(cl)`
ServerStats object for class *cl* at facility

`int stats.completions()`
number of completions for class *cl* at facility

`double stats.queueLength()`
: mean queue length for class *cl* at facility

`double stats.responseTime()`
: mean response time for class *cl* at facility

`double stats.serviceTime()`
: mean service time for class *cl* at facility

`double stats.throughput()`
: mean throughput rate for class *cl* at facility

`double stats.utilization()`
: utilization for class *cl* at facility

4.13. Status Report

To obtain a report on the status of all of the facilities in a model:

Prototype: `void status_facilities()`

Example: `model.status_facilities();`

This report lists each facility along with the number of servers, the number of servers that are busy, the number of processes waiting, the name and id of each process at a server, and the name and id of each process in the queue.

4. Facilities

5. Storages

A CSIM *storage* is a resource that can be partially allocated to a requesting process. A storage consists of a counter (to indicate the amount of available storage) and a queue for processes waiting to receive their requested allocation

Usage and queueing statistics are automatically maintained for each storage unit. These are "printed" whenever a *report* or a *report_storages* statement is executed (see section 19.3, *CSIM Report Output*, for details about the reports that are generated).

5.1. Declaring and Initializing Storage

A storage object is as follows:

Example: `Storage m_str;`

Before a storage can be used, the constructor must be invoked:

Prototype: `Storage(char *name, int size)`

Example: `m_str = new Storage("mem", 1000);`

A newly created storage is created with all of the "storage" available. Storages should be declared with global variables in the *sim (main)* process, prior to the beginning of the simulation part of the model. A storage must be initialized via the *new storage* statement before it can be used in any other statement.

5.2. Allocating from a Storage

The elements of a storage can be allocated to a requesting process.

Prototype: `void allocate(int amount)`

Example: `m_str.allocate(10);`

The amount of storage requested is compared with the amount of storage available at *m_str.* If the amount of available storage is sufficient, the amount available decreases by the requested amount and the requesting process continues. If the amount of available storage is not sufficient, the requesting process is suspended. When some of the storage elements are deallocated by some other process, the highest priority waiting processes are automatically allocated their requested storage amounts (as they can be accommodated), and they are allowed to continue. The list of waiting processes is searched in priority order until a request cannot be satisfied. In order to preserve priority order, a new request that would fit but would get in front of higher priority waiting requests will be queued.

5.3. Deallocating from a Storage Unit

To return storage elements to a storage, the *deallocate* procedure is used.

Prototype: `void deallocate(int amount)`

Example: `m_str.deallocate(10);`

If there are processes waiting, the highest waiting priority processes are examined. Those that will now fit have their requests satisfied and are allowed to continue. Executing a *deallocate* statement causes no simulated time to pass.

Caution: There is no check to insure that a process returns only the amount of storage that it had been previously allocated.

5.4. Producing Reports

Reports for storages are most often produced by calling the *report* function, which reports for all CSIM objects. Reports can be produced for all existing storages by calling the *report_storages* function. The report for a storage, as illustrated below, gives the name of the storage, the size (initial amount), the average allocation request, the utilization, the average time each request is "in" the storage, the average queue length, the average response time and the number of completed requests.

STORAGE SUMMARY

storage name	size	alloc amount	alloc count	dealloc amount	dealloc count	util	in-que length	in-queue time
store	120	33.6	100335	30.0	100333	0.751	7.94470	7.91818

5.5. Resetting a Storage Unit

In some cases, it is necessary to reset the statistics counters for a specific storage unit.

Prototype: `void reset()`

Example: `m_str.reset();`

Executing this statement does not affect the state of the storage. The *reset* and the *reset_storages* statements each call the *reset ()* method for all storage units in the model.

5.6. An Array of Storages

In an array of storages, each element of the array is an individual storage.

Example: `Storage m_strs[];`

A storage set must be constructed before the elements of the set can be used.

Example:
```
m_strs = new Storage[5];
for(int i = 0; i < 5; i++)
   m_strs[i] = new Storage("str"+i, 100);
```

The example declares and then initializes a set of five storages, each with 100 elements of storage available at the onset of operation. Each

individual unit of storage is given a unique (indexed) name. In the example, the first storage in the set is named *str0*, the second is named *str1*, and so on. The last storage is named *str4*. Similarly, the individual units of storage are accessed as elements of an array. All of the operations that apply to a storage also apply to the individual units of a storage set.

Individual elements of a storage set are accessed as shown in these examples:

Example:
```
m_strs[i].allocate(10);
```

5.7. Allocating Storage with a Time-out

Sometimes, processes cannot wait indefinitely to allocate the needed amount of storage. If such a process executes the *timed_allocate* function, then, if the requested amount of storage is not available, the process will be suspended until either the requested amount of storage becomes available or the time-out interval expires.

Prototype:
```
boolean timed_allocate(int amount,
       double timeout)
```

Example:
```
result = m-str.timed_allocate(10,100.0);
   if(result) . . .
```

The process must check the function value *(result)* to determine whether or not the requested storage was obtained. If the value *false* is returned, the process did not obtain any of the requested storage. If the value *true* is returned, then the process did obtain the requested storage.

5.8. Inspector Methods

These functions each return a statistic that describes some aspect of the usage of the specified storage.

Prototype:	Functional Value:
`String name()`	name of store
`int capacity()`	number of storages defined for store
`int available()`	number of storages currently available at store
`int qlength()`	number of processes currently waiting at store
`double busySum()`	sum of requested amounts from store
`int sumAllocs()`	time-weighted sum of requests for store
`int sumDeallocs()`	time-weighted sum of releases of store
`double busySum()`	busy time-weighted sum of amounts for store
`double waitingTime()`	waiting time weighted sum of amounts for store
`int allocCount()`	total number of requests for store
`int deallocCount()`	total number of completed requests for store
`double elapsedTime()`	time at store that is spanned by report
`double utilization()`	utilization
`double queueLength()`	average queue length

double	`respsonseTime()`	average response time

5.9. Reporting Storage Status

Prototype: `void status_storages()`

Example: `status_storages();`

The report will be written to the default output location or to the specified by *set_output_file* (see section 21.6, *Output File Selection*).

5. Storages

6. Buffers

A CSIM *buffer* is a resource that can store (hold) a number of tokens. The primary operations for a buffer are *put*, which places a number of tokens into the buffer, and *get,* which removes a number of tokens from the buffer. A buffer has a maximum capacity for holding tokens. A *get* operation stalls if there are too few tokens in the buffer, and a *put* operation stalls if there is not enough space (unused capacity) in the buffer.

A buffer consists of a counter (indicating the number of tokens in the buffer), and two queues: a *put-queue*, for processes waiting to complete a *put* operation, and a *get-queue*, for processes waiting to complete a *get* operation.

Usage and queueing statistics are automatically maintained for each buffer. These are "printed" whenever a *report* or *report_buffers* statement is executed (see section 19.3, *CSIM Report Output*, for details about the reports that are generated).

6.1. Declaring and Initializing a Buffer

A buffer object is as follows:

Example: `Buffer m_b;`

Before a buffer can be used, it must be initialized by calling the *buffer* function.

Prototype: `Buffer(String name, int size);`

Example: `m_b = new Buffer("b", 10);`

A newly created buffer is empty. Buffers should be declared and initialized as global variables in the *sim (main)* process, prior to beginning the simulation part of the model. A buffer must be initialized via the *new buffer* statement before it can be used in any other statement.

6.2. Putting Tokens into a Buffer

Tokens can be added to a buffer using the *put* operation.

Prototype: `void put(int amt);`

Example: `m_b.put(5);`

The number of tokens being *put* (the amount) is compared with the space remaining in the buffer (the maximum size minus the current amount). If the available space is less than or equal to space

remaining, the amount of the *put* is added to the current amount and the process doing the *buffer-put* operation continues. If the amount specified in the *put* call exceeds the space remaining, the process is placed in the *put-queue* and is then suspended. When some other process (or processes) removes (*gets*) tokens, the highest priority process in the *put-queue* is checked; if its *put* request can be accommodated, the *buffer-put* is done and the process resumes at the statement following the *buffer-put* statement. If other processes in the *put-queue* can be accommodated, they too are processed and allowed to proceed.

6.3. Getting Tokens from a Buffer

Tokens can be removed from a buffer using the *buffer-get* statement.

Prototype: `void get(int amt);`

Example: `m_b.get(4);`

The number of tokens being requested in a *get* (the amount) is compared to the number in the buffer. If the amount is less than or equal to the number in the buffer, the amount is subtracted and the process doing the *buffer-get* proceeds. If not, the process doing the *buffer-get* is placed in the *get-queue* and then suspended. When another process (or processes) adds (*puts*) tokens to the buffer, the highest priority process in the *get-queue* is checked; if its *get* request can be satisfied, the *buffer-get* is done, and the process resumes at the statement following the *buffer-get*. If other processes in the *get-queue* can be accommodated, they too are processed and allowed to proceed.

6.4. Producing Reports

Reports for buffers are most often produced by calling the report function, which reports for all CSIM objects. Reports can be produced for all existing buffers by calling the *report_buffers* function. The report for a buffer gives the name of the buffer, followed by two sets of statistics: one summarizing the *put* operations and one summarizing the *get* operations.

BUFFER SUMMARY

buffer name	size	get amt	get qlen	get resp	get count	put amt	put qlen	put resp	put count
buff	20	2.4	0.00000	0.00000	32	3.5	1.54545	0.60714	28

6.5. Resetting a Buffer

In some cases, it is necessary to reset the statistics counters for a specific buffer.

Prototype: `void reset()`

Example: `m_b.reset();`

Executing this statement does not affect the state of the buffer or its servers. The *reset* and the *reset_buffers* statements each call *Buffer.reset ()* for all buffers in the model.

6.6. Timed Operations for Buffers

Sometimes, processes cannot wait indefinitely to either *get* tokens from or *put* tokens into a buffer.

Prototype: `boolean timed_get(int amt, double timeout);`

Example:
```
result = m_b.timed_get(5, 100.0);
if(result) . . .
```

and

Prototype: `boolean timed_put(int amt, double timeout);`

Example:
```
result = m_b.timed_put(5, 100.0);
if(result) {. . .
```

The process must check the function value *(result)* to determine whether the operation timed out or completed. If the value *false* is returned, the process did not *get* or *put* the amount. If the value *true* is returned, then the process did complete the operation successfully.

6.7. Inspector Functions

These functions each return a statistic or counter value that describes some aspect of the operation of a buffer:

Prototype:	Function Value:
`int current()`	current number of tokens
`int size()`	capacity of buffer
`int get_total()`	total amount retrieved
`int put_total()`	total amount put
`int get_count()`	number of get's
`int put_count()`	number of put's
`double get_timeQueue()`	sum of get-queue lengths
`double put_timeQueue()`	sum of get-queue lengths
`String name()`	name of buffer
`int get_current_count()`	current get-queue length
`int put_current_count()`	current put-queue length

6.8. Reporting Buffer Status

Prototype: `void status_buffers();`

Example: `status_buffers();`

The report will be written to the default output location or to the location specified by *set_output_file* (see section 21.6, *Output File Selection*).

7. Events

Events are used to synchronize the operations of CSIM processes. An event exists in one of two states: *occurred* or *not occurred*. A process can change the state of an event or it can suspend its execution until an event has occurred. When a process is suspended, it can join a set of processes, all of which will be resumed when the event occurs. Or, it can join an ordered queue from which only one process is resumed for each occurrence of the event. An event is automatically reset to the *not occurred* state when all of the suspended processes that can proceed have done so.

Advanced features of events include the ability to create sets of events for which processes can wait and the ability for a process to bound its waiting time by specifying a time-out. Events can also be used to construct other synchronization mechanisms such as semaphores.

7.1. Declaring and Initializing an Event

An event is declared in a CSIM program as follows:

Example:

```
Event m_ev;
```

Before anevent can be used, it must be initialized by invoking the *new event* constructor.

Prototype: `Event(String name)`

Example: `m_ev = new Event("done");`

An event is initialized in the *not occurred* state. The event name is used only to identify the event in output reports and trace messages.

7.2. Waiting for an Event to Occur

A process waits for an event to occur by calling the *wait* function.

Prototype: `void untimed_wait()`

Example: `m_ev.untimed_wait();`

If the event is in the *occurred* state, control returns from the *untimed_wait* function immediately, and the event is changed to the *not occurred* state. If the event is in the *not occurred* state, the calling process is suspended from further execution and control will not return from the *untimed_wait* function until some other process sets this event. When the event is set, all waiting processes will be resumed and the event will be placed in the *not occurred* state.

Note: CSIM for C/C++ uses the *wait()* method rather than *untimed_wait()*. The *wait()* command cannot be used in CSIM for Java due to a conflict with a standard Java routine.

7.3. Waiting with a Time-Out

Sometimes a process must not be suspended indefinitely while waiting for an event to occur. If a process calls the *timed_wait* function, it will be suspended until either the event is set or the specified amount of time has passed.

Prototype: `boolean timed_wait(double timeout)`

Example:
```
result = m_ev.timed_wait(100.0);
  if (result) {
```

The calling process should check the functional value to determine the circumstances under which it was resumed. If the value *true* is returned, the process was activated because the event has occurred; if the value *false* is returned, the specified amount of time passed without the event being set.

7.4. Queueing for an Event to Occur

A process joins the ordered queue for an event by calling the *queue* function.

Prototype: `void untimed_queue()`

Example:
```
m_ev.untimed_queue();
```

This function behaves similarly to the *wait* function, except that each time the event is set, only one queued process is resumed. The queue is maintained in order of process priority, with processes having the same priority being ordered by time of insertion into the queue.

Note: CSIM for Java contains two identical *queue* methods, *queue()* and *untimed_queue()*. The *queue()* method is a relic from CSIM for C/C++. The *untimed_queue()* method is preferred in order to keep consistent syntax with the *untimed_wait()* method. Note that a *wait()* method exists in CSIM for C/C++, but does not exist in CSIM for Java due to a conflict with a standard Java routine.

7.5. Queueing with a Time-out

If a process calls the *timed_queue* function, it will be suspended until either the event is set a sufficient number of times for the process to be activated, or until the specified amount of time has passed.

Prototype: `boolean timed_queue(double timeout)`

Example:
```
result = m_ev.timed_queue(100.0);
if (result) {...
```

The calling process should check the functional value to determine the circumstances under which it was resumed. If the value *true* is returned, the process was activated because the event occurred; if the value *false* is returned, the specified amount of time passed without the process being activated by the event being set.

7.6. Setting an Event

A process can put an event into the *occurred* state by calling the *set* function.

Prototype: `void set()`

Example: `m_ev.set();`

Calling this function causes all waiting processes and one queued process to be resumed. If there are no waiting or queued processes, the event will be in the *occurred* state upon return from the *set* function. If there are waiting or queued processes, the event will be in the *not occurred* state upon return. No simulation time passes during these activities. Setting an event that is already in the *occurred* state has no effect.

7.7. Clearing an Event

A process can put an event into the *not occurred* state by calling the *clear* function.

Prototype: `void clear()`

Example: `m_ev.clear();`

Clearing an event happens in zero simulation time and no processes are in any way affected. Clearing an event that is already in the *not occurred* state has no effect.

7.8. Collecting and Reporting Statistics for Events

A set of statistics on usage can be collected for specified events. Statistics collection for an event is initiated by executing the *monitor()* method.

Prototype: `void monitor()`

Example: `m_ev.monitor();`

The standard report function automatically proceeds to "print" a report for each monitored event. This report is as follows:

EVENT SUMMARY

event name	number of queue vst	avg que length	avg time queued	number of wait vsts	avg wait length	avg time waiting	number of set ops
ev	10	0.99010	1.00000	50	4.95050	1.00000	10

A separate report for all of the monitored events is produced using the `report_events()` procedure, as follows:

Prototype: `void report_events()`

Example: `report_events();`

All of the events in an `event_set` (see below) can be monitored as well.

Prototype: `void monitor()`

Example: `m_evs.monitor();`

7.9. Resetting an Event

In some cases, it is necessary to reset the statistics counters for a specific event.

Prototype: `void reset()`

Example: `ev.reset();`

Executing this statement does not affect the state of the event. The *reset* and the *reset_events* statements each call *reset_event()* for all events in the model.

7.10. Event Sets

An event set is an array of related events for which some special operations are provided. An event set is declared as follows:

Example: `EventSet m_evSet;`

All events in an event set are initialized by invoking the *event_set* constructor.

Prototype: `EventSet(char *name, int numEvents)`

Example: `m_evSet = new EventSet("events", 10);`

As with any Java array, the events in an event set are indexed from *0* to *numEvents - 1*.

Individual events in the event set can be accessed and then manipulated using any of normal event functions (*e.g.*, *set*, *clear*, *untimed_wait*, *timed_wait*, *untimed_queue*, *timed_queue*).

Example:
```
Event e;
e = m_evSet.get(1);
e.set();
```

Example:
```
m_evSet.get(1).set();
```

A process can wait for the occurrence of any event in an event set by calling the *wait_any* function.

Prototype:
```
int wait_any()
```

Example:
```
eventIndex = evSet.wait_any();
```

This function returns the index of the event that caused the calling process to proceed. If multiple events in the set are in the *occurred* state, the lowest numbered event is the one recognized by the calling process. All processes that have called *wait_any* are activated by the next event that occurs, and these processes all receive the same index value.

A process can join an ordered queue for an event set by calling the *queue_any* method.

Prototype:
```
int queue_any()
```

Example:
```
eventIndex = m_evSet.queue_any();
```

Each time any event in the event set occurs, one process in the queue is activated. The functional value is the same as that of the *wait_any* function.

A process can also do a *timed_wait_any()* operation at an *event_set*.

Prototype:
```
int timed_wait_any(double time_out)
```

Example:
```
result = m_evSet.timed_wait_any(double
                    time_out);
```

If the result is not equal to *-1*, then it signifies the index of the event that caused the process to continue. If the result is equal to *-1*, then the operation timed out and no event was set.

A process can also do a *timed_queue_any()* operation at an *event_set*:

Prototype: `int timed_queue_any(double time_out)`

Example:
```
result = m_evSet.timed_queue_any(double
                    time_out);
```

If the result is not equal to *-1*, then it signifies the index of the event that caused the process to continue. If the result is equal to *-1*, then the operation timed out and no event was set.

7.11. Inspector Methods

The following methods return information about the specified event at the time they are called.

Prototype:	**Functional value:**
`String name()`	pointer to name of event
`int wait_cnt()`	number of processes waiting for event
`int queue_cnt()`	number of processes queued for event
`int qlength()`	sum of *wait_cnt* and *queue_cnt*
`int state()`	state of event: *OCC* if occurred or *NOT_OCC* if not occurred
`int num_events()`	number of processes waiting for the event
`int set_count()`	number of set operations
`double queue_sum()`	sum of queue queue lengths
`double wait_sum()`	sum of wait queue lengths

`int queue_delay_count()`	number of queue delays
`int wait_delay_count()`	number of wait delays
`double queue_length()`	average queue queue length
`double wait_length()`	average wait queue length
`double queue_time()`	average queue queue time
`double wait_time()`	average wait queue time
`int queue_count()`	number queue-queue visits
`int wait_count()`	number wait-queue visits

7.12. Status Report

The *status_events* function prints a report of the status of all events in the model.

Prototype: `void status_events()`

Example: `status_events();`

For each event, the report includes its state, the number of processes waiting for it, the number of processes queued for it, the name and id of all waiting processes, and the name and id of all queued processes. The report is written to the default output stream or the stream specified in the last call to *set_output_file*.

7.13. Built-In Events

A process can suspend itself until there are no other active processes by waiting on the built-in event *event_list_empty*.

Example: `eventListEmpty.untimed_wait();`

This event is automatically set by CSIM when all processes have terminated or are waiting for something (*e.g.*, a facility or storage). Modelers sometimes use this to force the initial (*sim*) process to wait until all work in the system being modeled has completed. Upon reactivation, the initial process might then produce reports.

If *run length control* is involved for a *table*, *qtable*, *meter,* or *box*, (see 14.3), a process can suspend itself until the run length control mechanism signals the end of a run, by waiting for the built-in event *converged*.

Example: `converged.untimed_wait()`

8. Mailboxes

A *mailbox* allows for the asynchronous exchange of data between CSIM processes. Any process may send a message to any mailbox, and any process may attempt to receive a message from any mailbox.

A mailbox is comprised of two FIFO queues: a queue of unreceived messages and a queue of waiting processes. At least one of the queues will be empty at any time. When a process sends a message, the message is given to a waiting process (if one exists) or it is placed in the message queue. When a process attempts to receive a message, it is either given a message from the message queue (if one exists) or it is added to the queue of waiting processes.

A message is a data object. If a process sends an object, it is the responsibility of that process to maintain the integrity of the data until it is received and processed.

8.1. Declaring and Initializing a Mailbox

A mailbox is declared in a CSIM program as follows:

Example:

```
Mailbox  m_mb;
```

Before a mailbox can be used, it must be initialized by invoking the *new mailbox* constructor.

Prototype: `Mailbox(String name)`

Example: `m_mb = new Mailbox("requests");`

A newly-created mailbox contains no messages. The mailbox name is used only to identify the mailbox in output reports and trace messages.

8.2. Sending a Message

A process sends a message by calling the *send* function.

Prototype: `void send(Object msg)`

Example: `m_mb.send(msg);`

If one or more processes are waiting on this mailbox, the process at the head of the process queue will resume execution and will be given this message. If no processes are waiting, this message will be appended to the tail of the message queue. No simulation time passes during this method call.

The *message* is an object. Normally, an instance of a user-defined class is instantiated and sent to the mailbox.

8.3. Receiving a Message

A process receives a message by calling the *receive* function.

Prototype: `Object receive()`

Example:
```
inMsg = m_md.receive();
```

If one or more messages (message objects) are queued at this mailbox, the calling process is given the message at the head of the queue and continues executing. If no messages are queued, the process is suspended from further execution and is added to the tail of the process queue for this mailbox.

8.4. Receiving a Message with a Time-out

Sometimes a process must not wait indefinitely to receive a message. If a process calls the *timed_receive* function, it will be suspended either until a message is received or until the specified amount of time has passed.

Prototype: `Object timed_receive(double timeout)`

Example:
```
result = m_mb.timed_receive(100.0);
if (result ! = null) {...
```

The calling process can check the functional value to determine the circumstances under which it was resumed. If the value returned is not equal to *null*, the process was activated because a message was received; if the value returned is equal to *null*, the specified amount of time passed without the process being activated by the receipt of a message.

8.5. Collecting and Reporting Statistics for Mailboxes

A set of statistics on the usage can be collected for specified mailboxes. Statistics collection for a mailbox is initiated by executing the *monitor* method.

Prototype: `void monitor()`

Example: `m_mb.monitor();`

The standard report function automatically proceeds to "print" a report for each monitored mailbox. This report is as follows:

MAILBOX SUMMARY

mailbox name	number of proc visits	process qlength	process rspTime	number of messages	message qlength	message rspTime
mb	100	0.18737	0.32295	100	1.03091	1.79461

A separate report for all monitored mailboxes is produced by the *report_mailboxes()* procedure:

Prototype: `void report_mailboxes()`

Example: `report_mailboxes();`

8.6. Resetting a Mailbox

In some cases, it is necessary to reset the statistics counters for a specific mailbox.

Prototype: `void reset()`

Example: `m_mb.reset();`

Executing this statement does not affect the state of the mailbox. The *reset* and *reset_mailboxes* statements each call *reset()* for all mailboxes in the model.

8.7. Inspector Methods

The following methods return information about the specified mailbox at the time they are called.

Prototype:	Functional value:
`String name()`	name of mailbox
`int msg_count()`	if positive, number of unreceived messages if negative, magnitude is number of waiting processes
`int queue_count()`	number of processes queued at mailbox
`double proc_sum()`	sum of process queue lengths
`int proc_delay_count()`	number of processes delayed

`int proc_count()`	number of processes receiving messages
`double proc_length()`	average process queue length
`double proc_time()`	average process delay time
`double msg_sum()`	sum of message queue lengths
`int msg_delay_count()`	number of messages delayed
`int msg_count()`	number of messages sent
`double msg_length()`	average message queue length
`double msg_time()`	average message delay time

8.8. Status Report

The *status_mailboxes* function prints a report of the status of all mailboxes in the model.

Prototype: `void status_mailboxes()`

Example: `status_mailboxes();`

For each mailbox, the report includes the number of unreceived messages, the number of waiting processes, and the name and id of all waiting processes. The report is written to the default output stream or the stream specified in the last call to *set_output_file.*

8. Mailboxes

9. Managing Queues

Each of the CSIM objects (*facilities*, *storages*, *buffers*, *events* and *mailboxes*) consists of one or more queues (for suspended processes) and some other structure (servers in a facility, a list of messages in a mailbox, etc.). In some models, it is necessary to be able to manipulate the processes in one of these queues. The *queue management* features are used to accomplish this.

9.1. Process Objects and Process Structures

The class for representing processes is *process*: This is the object returned when a process is requested from a CSIM object.

A process instance can be used to query the state of a process and to change attributes of a process.

Example: `Process p;`

Prototype: `int p.priority()`

Example: `pri = p.priority();`

Prototype: `void set_priority(int prty)`

Example: `p.set_priority(5);`

Prototype: `int identity()`

Example: `id = p.identity();`

Prototype: `Sting name()`

Example: `String nm = p.name();`

A process can have its own user-defined structure. This structure can be established by a process and be retrieved and examined by this process and other processes using a process pointer.

Prototype: `Object getProcessStructure();`

Example: `MyStruct ms = getProcessStructure();`

A process can establish its own private structure.

Prototype: `void setProcessStructure(Object ps)`

Example: `setProcessStructure(ms);`

A process can retrieve a pointer to the private structure for another process.

Prototype: `Object getProcessStructure()`

Example: `MyStruct ms = p.getProcessStructure();`

A process can set a private structure for another process.

Prototype: `void setProcessStructure(Object ps)`

Example: `p.setProcessStructure(ps);`

9.2. Process Queues at Facilities

The queue of processes at a facility can be managed by a set of routines.

To gain access to the first process in the process queue at a facility:

Prototype: `Process first_process()`

Example: `Process firstProcess = m_fac.first_process();`

If there are no processes in the process queue, the value returned is *null*; if there are processes in the process queue, the value returned is the first process in the queue.

Note: In these methods, when a process is retrieved, it is not removed from the queue; the *remove_process* methods do that.

The pointer to the last process at the end of the process queue at a facility (the process at the tail of the queue) can be retrieved as follows:

Prototype: `Process last_process()`

Example: `Process lstPrc = m_fac.last_process();`

If there are no processes in the process queue, the value returned is *null*; if there are processes in the process queue, the value returned is the last process in the queue.

A specific process can be removed from the process queue at a facility, as follows:

Prototype: `Process remove_process(Process pr)`

Example: `Process pr = m_fac.remove_process(pr1);`

The process at the head of the queue of waiting processes (the first process) can be removed from the process queue at a facility, as follows:

Prototype: `Process remove_first_process(Process pr)`

Example:
```
Process pr1 =
m_fac.remove_first_process(pr);
```

A process can be placed in a facility queue; the position of this process is determined by its priority relative to the other processes in the queue.

Prototype: `void insert_process(Process pr)`

Example: `m_fac.insert_process(pr);`

The *process_list()* method returns a linked list of the processes in the queue of waiting processes. This list is a “copy” of the queue. The methods of a Java object *List* can be used to access the elements (*Processes*) in this list.

Prototype: `List process_list();`

Example: `List pList = m_facility.process_list();`

9.3. Process Queues at Storages

The routines for managing processes in a storage queue are described in this section.

The first process (the process at the head of the queue) can be retrieved as follows:

Prototype: `Process first_process()`

Example: `Process p = m_str.first_process();`

The last process at the end of the queue (the process at the tail of the queue) can be retrieved as follows:

Prototype: `Process last_process()`

Example: `Process p = m_str.last_process();`

A specific process can be removed from a storage queue as follows:

Prototype: `Process remove_process(Process p)`

Example: `Process p1 = m_str.remove_process(p);`

A process can be placed in a storage queue; the position of this process is determined by its priority relative to the other processes in the queue.

Prototype: `void insert_process(Process p)`

Example: `m_str.insert_process(p);`

The *process_list()* method returns a linked list of the processes in the queue of waiting processes. This list is a "copy" of the queue. The methods of a Java object *List* can be used to access the elements (*Processes*) in this list.

Prototype: `List process_list();`

Example: `List pList = m_storage.process_list();`

9.4. Process Queues at Buffers

A buffer has two queues of processes: the queue of processes waiting for a *put* operation to complete and the queue of processes waiting for a *get* operation to complete. These two queues are referred to as the *put-queue* and the *get-queue* respectively.

The first process in the put-queue (the process at the head of the put-queue) can be retrieved as follows:

Prototype: `Process put_first_process()`

Example: `Process p = m_buf.put_first_process();`

The last process at the end of the put-queue (the process at the tail of the put-queue) can be retrieved as follows:

Prototype: `Process put_last_process()`

Example: `Process p = m_buf.put_last_process();`

A specific process can be removed from a put-queue queue as follows:

Prototype: `Process put_remove_process(Process p)`

Example: `Process p1 = m_buf.put_remove_process(p);`

A process can be placed in a buffer put-queue; the position of this process is determined by its priority relative to the other processes in the queue.

Prototype: `void put_insert_process(Process p)`

Example: `m_buf.put_insert_process(p);`

The *process_list()* method returns a linked list of the processes in the put-queue of waiting processes. This list is a "copy" of the queue. The methods of a Java object *List* can be used to access the elements (*Processes*) in this list.

Prototype: `List put_process_list();`

Example: `List pList = m_buf.put_process_list();`

The first process in the get-queue (the process at the head of the get-queue) can be retrieved as follows:

Prototype: `Process get_first_process()`

Example: `Process p = m_buf.get_first_process();`

The last process at the end of the get-queue (the process at the tail of the get-queue) can be retrieved as follows:

Prototype: `Process get_last_process()`

Example: `Process p = m_buf.get_last_process();`

A specific process can be removed from a get-queue as follows:

Prototype: `Process get_remove_process(Process p)`

Example: `Process p1 = m_buf.get_remove_process(p);`

A process can be placed in a buffer get-queue; the position of this process is determined by its priority relative to the other processes in the queue.

Prototype: `void get_insert_process(Process p)`

Example: `m_buf.get_insert_process(p);`

The *process_list()* method returns a linked list of the processes in the get-queue of waiting processes. This list is a “copy” of the queue. The methods of a Java object *List* can be used to access the elements (*Processes*) in this list.

Prototype: `List get_process_list();`

Example: `List pList = m_buf.get_process_list();`

9.5. Process Queues at Events

An event has two queues of processes: the queue of processes doing a *wait* operation and the queue of processes doing a *queue* operaton. These two queues are referred to as the *wait-queue* and the *queue-queue* respectively.

The first process of the wait-queue (the process at the head of the wait-queue) can be retrieved as follows:

Prototype: `Process first_wait_process()`

Example: `Process p = m_ev.first_wait_process();`

The last process at the end of the wait-queue (the process at the tail of the wait-queue) can be retrieved as follows:

Prototype: `Process last_wait_process()`

Example: `Process p = m_ev.last_wait_process();`

A specific process can be removed from a wait-queue queue as follows:

Prototype: `Process remove_wait_process(Process p)`

Example: `Process p = m_ev.remove_wait_process(p);`

A process can be placed in an event wait-queue; the position of this process is determined by its priority relative to the other processes in the queue.

Prototype: `void insert_wait_process(Process p)`

Example: `m_ev.insert_wait_process(p);`

The *process_list()* method returns a linked list of the processes in the *wait-queue* of waiting processes. This list is a “copy” of the queue. The methods of a Java object *List* can be used to access the elements (*Processes*) in this list.

Prototype: `List wait_process_list();`

Example: `List pList = m_ev.wait_process_list();`

The first process of the queue-queue (the process at the head of the queue-queue) can be retrieved as follows:

Prototype: `Process first_queue_process()`

Example: `Process p = m_ev.first_queue_process();`

The last process at the end of the queue-queue (the process at the tail of the queue-queue) can be retrieved as follows:

Prototype: `Process last_queue_process()`

Example: `Process p = m_ev.last_queue_process();`

A specific process can be removed from a queue-queue as follows:

Prototype: `Process remove_queue_process(Process p)`

Example: `Process p = m_ev.remove_queue_process(p);`

A process can be placed in a queue-queue; the position of this process is determined by its priority relative to the other processes in the queue.

Prototype: `void insert_queue_process(Process p)`

Example: `m_ev.insert_queue_process(p);`

The *process_list()* method returns a linked list of the processes in the queue-queue of waiting processes. This list is a “copy” of the queue. The methods of a Java object *List* can be used to access the elements (*Processes*) in this list.

Prototype: `List queue_process_list();`

Example: `List pList = m_ev.queue_process_list();`

9.6. Process Queues and Message Lists at Mailboxes

A CSIM mailbox consists of queue of processes waiting to receive messages and a list of messages waiting to be received by processes.

The first process in the *process queue* at a mailbox (the process at the head of the process-queue) can be retrieved as follows:

Prototype: `Process first_process()`

Example: `Process p = m_bx.first_process();`

The last process at the end of the process queue at a mailbox (the process at the tail of the process queue) can be retrieved as follows:

Prototype: `Process last_process()`

Example: `Process p = m_bx.last_process();`

A specific process can be removed from the process-queue at a mailbox as follows:

Prototype: `Process remove_process(Process p)`

Example: `Process p1 = m_bx.remove_process(p);`

A process can be placed in the process queue at a mailbox; the position of this process is determined by its priority relative to the other processes in the queue.

Prototype: `void insert_process(Process p)`

Example: `m_bx.insert_process(p);`

Assume that a class named *Msg_c* has been defined.

The first message of the *message list* at a mailbox can be retrieved as follows:

Prototype: `Object first_msg()`

Example: `Msg_c m = (Msg_c)m_bx.first_msg();`

The last message at the end of the message list at a mailbox can be retrieved as follows:

Prototype: `Object last_msg()`

Example: `Msg_c m = (Msg_c)m_bx.last_msg();`

A specific message can be removed from the message list at a mailbox as follows:

Prototype: `Object remove_message(Object m)`

Example: `Msg_c m1 = (Msg_c)m_bx.remove_message(m);`

A message can be placed in the message list at a mailbox; the message will be placed at the end of the message list.

Prototype: `void insert_message(Object o)`

Example: `m_b.insert_message(m);`

An instance of a *message* class is a *message* object. A pointer to a *message* object is defined as follows:

Example: `message *mptr;`

Example: `message_t mptr;`

9. Managing Queues

10. Introduction to Statistics Gathering

CSIM automatically gathers and reports performance statistics for certain types of model components, including *facilities* and *storages*. CSIM also provides four general-purpose statistics gathering tools: *tables, qtables, meters*, and *boxes.* These tools can be used for the following purposes:

- To obtain statistics other than mean values for *facilities* and *storages*
- To obtain statistics on processes
- To obtain statistics for selected submodels or for the model considered as a whole
- To obtain confidence intervals for selected statistics
- To employ the run length control algorithms provided with CSIM (see section 16.3, *Run Length Control*)

Of course, any statistics can be gathered by declaring and updating variables in a CSIM program. But the statistics gathering tools are powerful and comprehensive, and their use will decrease the likelihood of programming errors that lead to incorrect statistics. Formatted reports of the statistics gathered with these tools can easily be included in the model output.

The following steps are suggested for adding statistics gathering to a model:

- Identify what statistics are of interest and which statistics gathering tools are appropriate.
- Declare a globally accessible object for each statistics gathering tool that will be used.

- Initialize each statistics gathering tool, usually at the beginning of the *sim* function.
- Add instrumentation (*i.e.*, function calls) to the model to feed data to the tools.
- Generate reports by calling the *report* function.

The magnitudes of the performance statistics obviously depend on the time unit that is chosen for the model. Most of the reports produced by the statistics gathering tools will accommodate floating point numbers with six digits to the left of the decimal point and six digits to the right of the decimal point. Up to nine digits can be displayed for integer values. The time unit should be chosen to avoid performance values so far from unity that digits of interest are not displayed.

11. Tables

A *table* is used to gather statistics on a sequence of discrete values such as interarrival times, service times, or response times. Data values are “recorded” in a table to include them in the statistics. A table does not actually store the recorded values; it simply updates the statistics each time a value is included.

The statistics maintained by a table include the minimum, maximum, range, mean, variance, standard deviation, and coefficient of variation. Optional features for a table allow the creation of a histogram, the calculation of confidence intervals, and the computation of statistics for values in a moving window.

First-time users of tables should focus on the following three sections, which explain how to set up tables, record values, and produce reports. Subsequent sections describe the more advanced features of tables.

11.1. Declaring and Initializing a Table

A table is declared in a CSIM program as follows:

Example:
```
Table tbl;
```

Before a table can be used, it must be initialized by invoking the *table* constructor:

Prototype: `Table(char* name);`

Example: `Table tbl = Table("response times");`

The table name is used only to identify the table in the output reports. Up to 80 characters in the name will be stored by CSIM. A newly created table contains no values and all the statistics are zero.

A table can be designated as a *permanent table* using the *setPermanent* function.

Prototype: `setPermanent(boolean t)`

Example: `tbl.setPermanent(true);`

The information in a permanent table is not cleared when the *reset* function is called, and a permanent table is not deleted when *rerun* is called. In all other ways, a permanent table is exactly like any other table. Permanent tables are often used to gather data across multiple runs of a model. As a general rule, do not make a table permanent unless you have a specific reason for doing so.

11.2. Tabulating Values

A value is included in a table using the *tabulate* function.

Prototype: `void tabulate(double value)`

Example: `tbl.tabulate(1.0);`

Tables are designed to maintain statistics on data of type *double*. Data of other types, such as *integer*, must be cast to type *double* in the call to record.

The *record()* method is equivalent to the *tabulate()* method:

Prototype: `void record(double value)`

Example: `tbl.record(1.0);`

11.3. Producing Reports

Reports for tables are most often produced by calling the *report* procedure, which prints reports for all statistics gathering objects.

Reports can be produced for all existing tables by calling the *report_tables* method; this method is part of the *Model* object.

Prototype: `void report_tables()`

Example: `model.report_tables();`

The report for a table will include the table name and all statistics, as illustrated below. If the table is empty, a message to that effect is printed instead of the statistics.

TABLE 1: response times			
minimum	0.009880	mean	2.881970
maximum	13.702809	variance	7.002668
range	13.692929	standard deviation	2.646255
observations	962	coefficient of var	0.918211

11.4. Histograms

A *histogram* can be specified for a table in order to obtain more detailed information about the recorded values. The mode and other percentiles can often be estimated from a histogram. A histogram is specified for a table by calling the *add_histogram* function.

Prototype:
```
void add_histogram(long nbucket,
  double min, double max)
```

Example:
```
tbl.add_histogram(10, 0.0, 10.0);
```

The number of buckets in the histogram will be *nbucket*. The smallest value in the first bucket will be *min*; the largest value in the last bucket will be *max*. All buckets will have the same width of (*max-min*)/*nbucket*. An underflow bucket and an overflow bucket will automatically be created if needed to hold values less than *min* or greater than *max*. The index of the underflow bucket is *0*, and the index of the overflow bucket is *nbucket+1*.

Usually, a histogram is specified for a table immediately after the table is initialized. Additional calls can be made to *add_histogram* to change the characteristics of the histogram, but only if the table is empty.

A report for a table that has a histogram will include an additional section, as illustrated below. For each bucket in the histogram, the following information will be displayed: the smallest value the bucket can hold, the number of values in the bucket, the proportion of all values that are in the bucket, the proportion of all values in the bucket and all preceding buckets, and a bar whose length corresponds to the proportion of values in the bucket.

lower limit	frequency	proportion	cumulative proportion	
0.00000	265	0.275468	0.275468	**********
1.00000	219	0.227651	0.503119	********
2.00000	125	0.129938	0.633056	*****
3.00000	92	0.095634	0.728690	****
4.00000	74	0.076923	0.805613	****
5.00000	54	0.056133	0.861746	***
6.00000	53	0.055094	0.916840	***
7.00000	38	0.039501	0.956341	**
8.00000	8	0.008316	0.964657	*
9.00000	8	0.008316	0.972973	*
>=10.00000	26	0.027027	1.000000	**

If leading or trailing buckets contain no values, the lines in the report for these buckets will not be printed. This feature allows the histogram to be output as compactly as possible without losing any information.

CSIM must save information for each bucket in a histogram. Consequently, the storage requirements for a table that has a histogram are proportional to the number of buckets.

11.5. Confidence Intervals

CSIM can automatically compute *confidence intervals* for the mean of the data in any table. The confidence interval calculations are enabled by calling the *confidence* function.

Prototype: `void confidence()`

Example: `tbl.confidence();`

If confidence intervals have been requested, the report for a table will have an additional section, as illustrated below.

confidence intervals for the mean after 50000 observations

level	confidence interval	rel. error
90 %	4.114119 +/- 0.296434 = [3.817684, 4.410553]	0.077648
95 %	4.114119 +/- 0.354041 = [3.760078, 4.468159]	0.078837
98 %	4.114119 +/- 0.421555 = [3.692563, 4.535674]	0.080279

Chapter 16, *Confidence Intervals and Run Length Control*, describes confidence intervals in detail and explains how to interpret the information in this report.

11.6. Inspector Methods

All statistics maintained by a table can be retrieved during the execution of a model or upon its completion. The attributes of a table (*i.e.*, its name and moving window size) can also be retrieved.

Prototype:	Functional value:
`String name()`	name of table
`int countt()`	number of values recorded
`double min()`	minimum value
`double max()`	maximum value
`double sum()`	sum of values
`double sumSquares()`	sum of squares of values
`double mean()`	mean of values
`double range()`	range of values
`double var()`	variance of values
`double stddev()`	standard deviation of values
`double cv()`	coefficient of variation of values
`double elapsedTime()`	time since last reset
`Histogram histogram()`	histogram object in table
`ConfidenceInterval getConfidenceInterval`	confidence interval object in table

The following inspector methods retrieve information about the histogram associated with a table:

Prototype:	Functional value:
`double histogram_width()`	width of each bucket
`int histogram_bucket(int n)`	contents of bucket *n*
`double histogram_high()`	value of *max* bucket
`double histogram_low()`	value of *min* bucket
`double histogram_num()`	number of buckets
`double histogram_total()`	sum of bucket contents

Note: The number of buckets in a histogram does not include the underflow or overflow buckets. Bucket number *0* is the underflow bucket; bucket number *1+histogram_num*() is the overflow bucket. If a histogram has not been specified for a table, the above inspector methods all return zero values.

The following inspector methods retrieve information about the confidence interval associated with a table:

Prototype:	Functional value:
`double conf_halfwidth (double level)`	half width
`double conf_lower (double level)`	lower end
`double conf_upper(double level)`	upper end

The following inspector methods retrieve information about the run length control associated with a table:

Prototype:	Functional value:

`int batch_size()`	current size of batch
`int batch_count()`	number of batches used
`boolean converged()`	TRUE or FALSE
`double table::conf_mean()`	mid point of conf. int.
`double table::conf_accuracy (double level)`	accuracy achieved

Although most statistics are mathematically undefined if there is no data, the corresponding inspector methods return a value of zero if the table is empty.

The inspector methods that retrieve information about the results of run length control are described in section 16.3, *Run Length Control*.

11.7. Resetting a Table

Resetting a table causes all information maintained by the table to be reinitialized. All optional features selected for the table (*e.g.*, histogram, confidence intervals, moving window) remain in effect and are also reinitialized.

The *reset* function is usually used to reset all statistics gathering tools at once. A specific table can be reset using the *reset_table* function.

Prototype: `void reset()`

Static Example:
```
tbl.reset();
```

Although permanent tables are not reset by the *reset* function, they can be reset explicitly by calling *reset_table*.

11. Tables

12. Qtables

A *qtable* is used to gather statistics on an integer-valued function of time, such as the length of a queue, the population of a subsystem, or the number of available resources. Every change in the value of the function must be "noted" by calling a CSIM function. A qtable does not actually save the functional values; it simply updates the statistics each time the value changes. (See section 12.6 for the only exception to this rule.)

The statistics maintained by a qtable include the minimum, maximum, range, mean, variance, standard deviation, and coefficient of variation. The number of changes in the functional value is maintained, as well as the initial and final values. Optional features for a qtable allow the creation of a histogram and the calculation of confidence intervals.

First-time users of qtables should focus on the following three sections, which explain how to set up qtables, note changes in their values, and produce reports. Subsequent sections describe the more advanced features of qtables.

12.1. Declaring and Initializing a Qtable

A qtable is declared in a CSIM program as follows:

Example: `QTable m_qtbl;`

Before a qtable can be used, it must be initialized by invoking the *qtable* constructor.

Prototype: `QTable(char *name)`

Example: `m_qtbl = new QTable("queue length");`

The qtable name is used only to identify the qtable in the output reports. CSIM will store up to 80 characters in the name. A newly created qtable has an initial value of zero. To create a qtable with a non-zero initial value, call the *note_value* method (described below) immediately after creating the qtable.

A qtable can be designated as a permanent qtable using the *setPermanent* method.

Prototype: `setPermanent(boolean t)`

Example: `m_qtbl. setPermanent(true);`

12.2. Noting a Change in Value

The most common way for the value of a qtable to change is for it to increase or decrease by one. Such a change would occur when a customer joins a queue or a resource is allocated. The value of a qtable is increased by one using the *note_entry* method.

Prototype: `void note_entry()`

Example: `m_qtbl.note_entry();`

The value of a qtable is decreased by one using the *note_exit* method.

Prototype: `void note_exit()`

Example: `m_qtbl.note_exit();`

The value of a qtable can be changed to an arbitrary number using the *note_value* method.

Prototype: `void note_value(int value)`

Example: `m_qtbl.note_value(12);`

12.3. Producing Reports

Reports for qtables are most often produced by calling the *report* function, which prints reports for all statistics gathering objects.

Reports can be produced for all existing qtables by calling the *report_qtables* function.

Prototype: `void report_qtables()`

Example: `report_qtables();`

The report for a qtable will include the qtable name and all statistics, as illustrated below. If no time has passed since the creation or reset of the qtable, a message to that effect is printed instead of the statistics.

QTABLE 1: queue length

initial	0	minimum	0	mean	2.788416
final	4	maximum	14	variance	8.529951
entries	966	range	14	standard deviation	2.920608
exits	962			coeff of variation	1.047408

12.4. Histograms

A histogram can be specified for a qtable in order to obtain more detailed information about the functional values. Depending on how the qtable is being used, its histogram might give the distribution of the queue lengths, the subsystem population, or the number of available resources. A histogram is specified for a table by calling the *add_histogram* method.

Prototype: `void add_histogram(long nbucket, long min, long max)`

Example: `m_qtbl.add_histogram(10, 0, 10);`

The number of buckets in the histogram will be *nbucket*. The smallest value in the first bucket will be *min*; the largest value in the last bucket will be *max.* All buckets will have the same width, which will be rounded up to an integer if necessary. An underflow bucket and an overflow bucket will automatically be created if needed to hold values less than *min* or greater than *max.*

Caution: The *min* and *max* parameters of *qtable_histogram* are of type *int*, whereas the analogous parameters of *table_histogram* are of type *double*.

Usually, a histogram is specified for a qtable immediately after the qtable is initialized. Additional calls can be made to *histogram* to

change the characteristics of the histogram, but only if the qtable is empty.

A report for a qtable that has a histogram will include an additional section, as illustrated below. For each bucket in the histogram, the following information will be displayed: the smallest value the bucket can hold, the total time the functional value was in the bucket, the proportion of time that the functional value was in the bucket, the proportion of all functional values in the bucket and all preceding buckets, and a bar whose length corresponds to the proportion of time the functional value was in the bucket.

number	total time	proportion	cumulative proportion	
0	248.74145	0.249003	0.249003	**********
1	185.45534	0.185651	0.434654	*******
2	157.13503	0.157300	0.591954	******
3	100.01937	0.100125	0.692079	****
4	78.14196	0.078224	0.770303	***
5	62.59210	0.062658	0.832961	***
6	44.38455	0.044431	0.877392	**
7	35.33308	0.035370	0.912762	**
8	25.94494	0.025972	0.938735	*
9	21.48465	0.021507	0.960242	*
>= 10	39.71625	0.039758	1.000000	***

If leading or trailing buckets contain no values, the lines in the report for these buckets will not be printed. This feature allows the histogram to be output as compactly as possible without losing any information.

CSIM must save information for each bucket in a histogram. Consequently, the storage requirements for a qtable that has a histogram are proportional to the number of buckets.

12.5. Confidence Intervals

CSIM can automatically compute *confidence intervals* for the mean value of any qtable. The confidence interval calculations are enabled by calling the *confidence* method.

Prototype: `void confidence()`

Example: `m_qtbl.confidence();`

If confidence intervals have been requested, the report for a qtable will include an additional section, as illustrated below.

confidence intervals for the mean after 29600.000000 time units

level	confidence interval	rel. error
90 %	4.319412 +/- 0.491696 = [3.827715, 4.811108]	0.128457
95 %	4.319412 +/- 0.588209 = [3.731203, 4.907621]	0.157646
98 %	4.319412 +/- 0.701971 = [3.617441, 5.021382]	0.194052

Section 16.1, *Confidence Intervals*, describes confidence intervals in detail and explains how to interpret the information in this report.

12.6. Inspector Methods

All statistics maintained by a qtable can be retrieved during the execution of a model or upon its completion. The attributes of a qtable (*i.e.*, its name and moving window size) can also be retrieved.

Prototype: **Functional value:**

`String name()`	name of qtable
`int entries()`	number of *note_entry*'s
`int exits()`	number of *note_exit*'s
`int min()`	minimum value
`int max()`	maximum value
`int current()`	current value
`double sum()`	sum of values weighted by time
`double sumSquares()`	sum of squared weighted
`double mean()`	mean value
`int range()`	range of values
`double var()`	variance of values
`double stddev()`	standard deviation of values
`double cv()`	coefficient of variation of values

The following inspector methods retrieve information about the confidence interval associated with a table:

Prototype:	**Functional value:**
`double conf_mean(double level)`	half width
`double conf_halfwidth(double level)`	half width
`double conf_lower(double level)`	lower end
`double conf_upper(double level)`	upper end

The following inspector methods retrieve information about the run length control associated with a table:

Prototype:	**Functional value:**
`int batch_size()`	current size of batch

`int batch_count()`	number of batches used
`boolean converged()`	TRUE or FALSE
`double conf_mean()`	mid point of conf. int.
`double conf_accuracy (double level)`	accuracy achieved

Many statistics are mathematically undefined if zero time has passed since the creation or reset of a qtable. The corresponding inspector methods return a value of zero in this case.

The following inspector methods retrieve information about the histogram associated with a qtable.

Prototype:	**Functional value:**
`int hist_num()`	number of buckets
`double hist_low()`	smallest value that is not underflow
`double hist_high()`	largest value that is not overflow
`double hist_width()`	width of each bucket
`int hist_bucket(int i)`	total time value is in bucket *i*

The number of buckets in a histogram does not include the underflow or overflow buckets. Bucket number *0* is the underflow bucket; bucket number *1+histogram_num*() is the overflow bucket. If a histogram has not been specified for a qtable, the above inspector methods all return zero values.

The inspector methods that retrieve information about the results of run length control are described in section 16.3, *Run Length Control*.

12.7. Resetting a Qtable

Resetting a qtable causes all information maintained by the qtable to be reinitialized, except that the current value is saved for use in computing future values. All optional features selected for the qtable (*e.g.,* histogram, confidence intervals, moving window) remain in effect and are also reinitialized.

The *reset* function is usually used to reset all statistics gathering tools at once. A specific qtable can be reset using the *reset* method.

Prototype: `void reset()`

Example: `m_qtbl.reset();`

Although permanent qtables are not reset by the *reset* function, they can be reset explicitly by calling *reset_qtable*.

12. Qtables

13. Meters

A *meter* is used to gather statistics on the flow of entities, such as customers or resources, past a specific point in a model. Meters can be used to measure arrival rates, completion rates, and allocation rates. A meter can be thought of as a probe that is inserted at some point in a model.

While a meter primarily measures the rate at which entities flow past it, a meter also keeps statistics on the times between passages. These interpassage times are recorded in a table, which is an integral part of every meter.

First-time users of meters should focus on the following three sections, which explain how to set up meters, update meters, and produce reports. Subsequent sections describe the more advanced features of meters.

13.1. Declaring and Initializing a Meter

A meter is declared in a CSIM program as follows:

Example:

```
Meter m_mtr;
```

Before a meter can be used, it must be initialized by invoking the *meter* constructor.

Prototype: `Meter(String name)`

Example: `m_mtr = new Meter("system completions");`

The meter name is used only to identify the meter in the output reports. CSIM will store up to 80 characters in the name.

13.2. Instrumenting a Model

An entity notes its passage by a meter using the *note_passage* method.

Prototype: `void note_passage()`

Example: `m_mtr.note_passage();`

For the statistics to be accurate, every entity of interest must note its passage and do so at the correct time.

13.3. Producing Reports

Reports for meters are most often produced by calling the *report* function, which prints reports for all statistics gathering objects.

Reports can be produced for all existing meters by calling the *report_meters* function.

Prototype: `void report_meters()`

Example: `report_meters();`

The report for a meter, as illustrated below, will include the meter name, the number of passages, the passage rate, and statistics on the interpassage times. If no time has elapsed, a message to that effect is printed instead of the statistics.

METER 2: System completions

count	494	rate	0.988000

interpassage time statistics

minimum	0.001258	mean	1.008764
maximum	6.533026	variance	0.994894
range	6.531768	standard deviation	0.997444
observations	494	coefficient of var	0.988778

13.4. Histograms

A histogram can be specified for the interpassage times of a meter. This is accomplished using the *add_histogram* method.

Prototype: `void add_histogram(long nbuckets, double min, double max)`

Example: `m_mtr.add_histogram(10, 0.0, 10.0);`

The histogram for a meter is exactly the same as the histogram for a table. See section 11.4, *Histograms*, for details.

13.5. Confidence Intervals

CSIM can automatically compute confidence intervals for the mean interpassage time at a meter. The confidence interval calculations are enabled by calling the *confidence* method.

Prototype: `void confidence()`

Example: `m_mtr.confidence();`

The confidence intervals for a meter are the same as the confidence intervals for a table. See section 16.1, *Confidence Intervals*, for details.

13.6. Inspector Methods

All statistics maintained by a meter can be retrieved during the execution of a model or upon its completion. The name of a meter can also be retrieved.

Prototype:	**Functional value:**
`String name()`	name of meter
`double elapsedTime()`	time spanned by data in interpassage time
`int cnt()`	number of passages noted
`double rate()`	rate of passages
`BasicTable ip_table()`	handle to interpassage time table

Although the passage rate is mathematically undefined if no time has passed, the *rate* method returns the value zero in this case.

The handle to a meter's interpassage time table can be passed to the inspector functions for a table in order to obtain interpassage time statistics.

Example: `double max_ip_time = m_mtr.ip_table().max();`

If no passages have occurred, the interpassage time table is empty. The interpassage time contributed by the first passage is the time from the beginning of the observation period to that first passage.

13.7. Resetting a Meter

Resetting a meter causes all information maintained by the meter to be reinitialized, except that the time of the last passage is saved for use in computing the next interpassage time. All optional features selected for the meter (*e.g.*, histogram, confidence intervals, moving window) remain in effect and are also reinitialized.

The *reset* function is usually used to reset all statistics gathering tools at once. A specific meter can be reset using the *reset_meter* function.

Prototype: `void reset()`

Example: `m_mtr.reset();`

14. Boxes

A *box* conceptually encloses part or all of a model. The box gathers statistics on the number of entities in the box (*i.e.*, the population) and the amount of time entities spend in the box (*i.e.*, the elapsed time). An *entity* might be a customer, a message, or a resource. Boxes are usually used to gather statistics on queue lengths, response times, and populations. Instrumenting a model involves inserting function calls at the places that entities enter and exit the box.

A table and a qtable are invisible but integral parts of every box. Statistics on the elapsed times are kept in the table, while statistics on the population are kept in the qtable.

First-time users of boxes should focus on the following three sections, which explain how to set up boxes, instrument a model, and produce reports. Subsequent sections describe the more advanced features of boxes.

14.1. Declaring and Initializing a Box

A box is declared in a CSIM program as follows:

Example:
```
Box  m_bx;
```

Before a box can be used, it must be initialized by invoking the *box* constructor.

Prototype: `Box(String name)`

Example: `m_bx = new Box("in system");`

The box name is used only to identify the box in the output reports. CSIM will store up to 80 characters in the name. A newly created box is always empty. To create a non-empty box, call the *enter* method (described in the following section) the appropriate number of times immediately after creating the box.

14.2. Instrumenting a Model

An entity enters a box by calling the *enter* method.

Prototype: `double enter()`

Example: `double entryTime = m_bx.enter();`

This function returns a timestamp that must be saved by the entity that entered the box. The entity exits the box by calling the *exit* method and passing to it the timestamp that it received upon entry.

Prototype: `void exit(double et);`

Example: `m_bx.exit(entryTime);`

It is the responsibility of the programmer to ensure that the integrity of the timestamp is maintained while the entity is in the box. Because boxes may be nested or may overlap, it is advisable to make the timestamp local to the CSIM process and to use a separate timestamp variable for each box. An invalid timestamp (*i.e.*, one that is less than zero or greater than the current time) will cause an error.

14.3. Producing Reports

Reports for boxes are most often produced by calling the *report* function, which prints reports for all statistics gathering objects.

Reports can be produced for all existing boxes by calling the *report_boxes* function.

Prototype: `void report_boxes()`

Example: `report_boxes();`

The report for a box, as illustrated below, will include the box name, statistics on the elapsed times, and statistics on the population of the box. If the box is empty or no time has passed since its creation or reset, messages to that effect are printed instead of the statistics. Note that statistics on the elapsed times reflect only those entities that have exited the box. Entities still in the box when the report is produced contribute to the population statistics but not to the elapsed time statistics.

BOX 1: Queue statistics

statistics on elapsed times

minimum	0.009880	mean	2.088345
maximum	7.943915	variance	3.211423
range	7.934035	standard deviation	1.792044
observations	494	coefficient of var	0.858117

statistics on population

initial	0	minimum	0	mean
final	7	maximum	10	variance
entries	501	range	10	standard deviation
exits	494			coeff of variation

14.4. Histograms

A histogram can be specified for the elapsed times in a box and for the population of a box using the following functions:

Prototype: `void add_time_histogram(int nb, double min, double max)`

Example: `m_bx.add_time_histogram (10, 0.0, 10.0);`

Prototype: `void add_number_histogram(int nb, long min, long max)`

Example: `m_bx.add_number_histogram(10, 0, 10);`

The histogram for the elapsed times is the same as the histogram for a table. See section 11.4, *Histograms*, for details. The histogram for the population of a box is the same as the histogram for a qtable. See section 12.4, *Histograms*, for details

Caution: The *min* and *max* parameters of *box_time_histogram* are of type *double*, whereas the corresponding parameters of *box_number_histogram* are of type *long*.

14.5. Confidence Intervals

Confidence intervals can be requested for the mean of the elapsed times in a box and for the mean population of a box using the following functions:

Prototype: `void time_confidence()`

Example: `m_bx.time_confidence();`

Prototype: `void number_confidence()`

Example: `m_bx.number_confidence();`

These two types of confidence intervals are identical to the confidence intervals for a table and qtable, respectively. See section 16.1, *Confidence Intervals*, for details.

14.6. Inspector Methods

All statistics maintained by a box can be retrieved during the execution of a model or upon its completion. The name of a box can also be retrieved.

Prototype:	**Functional value:**
`String name`	name of box
`BasicTable time_table`	handle to elapsed time table
`BasicQTable box::number_qtable`	handle to population qtable

The handle to a box's elapsed time table can be passed to the inspector methods for a table in order to obtain statistics on the times that entities have spent in the box.

Example:
```
double max_time_in_box =
m_bx.time_table().max();
```

If no entities have exited the box, the table will be empty and zeros will be returned for the undefined statistics.

The handle to a box's population qtable can be passed to the inspector methods for a qtable in order to obtain statistics on the population.

Example:
```
int max_population =
m_bx.number_qtable->max();
```

If no time has passed, zero values will be returned for the undefined statistics.

14.7. Resetting a Box

Resetting a box causes all information maintained by the box to be reinitialized, except that the number currently present in the box is saved for use in computing future populations. All optional features selected for the box (*e.g.*, histogram, confidence intervals, moving window) remain in effect and are also reinitialized.

The *reset* function is usually used to reset all statistics gathering tools at once. A specific box can be reset using the *reset* method.

Prototype: `void reset()`

Example:
```
m_bx.reset();
```

15. Advanced Statistics Gathering

This section describes CSIM for Java's advanced statistics gathering features.

15.1. Example: Instrumenting a Facility

For each facility, CSIM automatically gathers and reports the following statistics:

- mean service time
- mean queue length
- utilization
- mean response time
- throughput
- number of completions

Meters and boxes can easily be used to gather more detailed statistics. The following statements show the declaration of the needed variables:

```
FCFSFacility  f;
Meter         arrivals;
Meter         departures;
Box           queue_box;
Box           service_box;
```

The following statements, which would normally appear in the *sim* process, show the initialization of the variables:

```
f = new FCFSFacility("center");
arrivals = new Meter("arrivals");
departures = new Meter("completions");
queue_box = new Box("queue");
service_box = new Box("in service");
```

The following code shows the instrumentation of the facility:

```
private class Customer() extends Process {
  public Customer() {
     super(“Customer”();
  }
  public void run() {
     double timestamp1;
     double timestamp2;

     arrivals.note_passage();
     timestamp1 = queue_box.enter();
     f.reserve();
     timestamp2 = service_box.enter();
     hold (rand.exponential(0.8));
     f.release();
     service_box.exit(timestamp2);
     queue_box.exit(timestamp1);
     departures.note_passage();
  }
}
```

The report for the box, *queue_box,* would give statistics on response times (under the heading, *statistics on elapsed times*) and queue lengths (under the heading, *statistics on population*). The report for the box, *service_box*, would give statistics on service times (under the heading, *statistics on elapsed times*) and utilization (under the heading, *statistics on population*). The report for the *arrivals* meter

would give statistics on the arrival rate and interarrival times. The report for the *departures* meter would give statistics on the completion rate and intercompletion times. If the arrival and completion rates were sufficiently similar, this quantity would be called the *throughput*.

Obviously, histograms could be added to any of these meters and boxes to obtain information on the various distributions.

15.2. The Report Function

Although reports can be produced at any time for individual statistics gathering tools, it is most common to generate reports for all tools at the same time, usually when the simulation has converged. This can be done by calling the *report* method (in the *Model* object).

Prototype `report()`

Example: `model.report();`

The *report* function produces reports for all facilities, storages, and classes, followed by reports for all tables, qtables, meters, and boxes. The sequence of reports begins with a header that includes the model name, the date and time, the current simulation time, and the CPU time used.

15.3. Resetting Statistics

CSIM provides a single function that will clear all accumulated statistics without affecting the state of the system being modeled in

any way. This *reset* method in the *Model* object is most often used when warming up a simulation. The simulation is begun with the system in an empty state, simply as a matter of convenience. A small number of customers is allowed to pass through the system, hopefully taking the system closer to its equilibrium state. Then, the statistics are reset and the simulation is run until convergence is achieved.

The *reset* function has a simple interface.

Prototype: `void reset()`

Example: `model.reset();`

Reset clears the statistics that are automatically gathered for facilities, storages, events, and process classes. It also resets the statistics in all non-permanent tables, qtables, meters, and boxes being used in the program. Permanent tables are not affected by calling *reset*.

In general, resetting statistics returns all the statistical counters and timers maintained by CSIM to their initial values, which are usually zero. But, there are a few subtle and important exceptions to this rule. When a qtable is reset, it remembers the current value for use in computing future values from the relative changes specified by *note_entry* and *note_exit*. When a meter is reset, it remembers the time of the last passage for use in computing the next interpassage time. When a box is reset, it remembers the number present for use in computing future populations.

Calling *reset* in no way changes the state of the system being modeled. It does not change the simulation clock; it does not affect the streams of random numbers being used in the simulation; and it does not affect the states of processes, facilities, storages, events, and mailboxes. The *reset* function is normally called during a simulation run, whereas the *rerun* function (see section 21.3.1, *To Rerun a CSIM Model*) is called between successive runs.

16. Confidence Intervals and Run Length Control

Most simulations are designed so they converge to what might be called the "true solution" of the model. But, because a simulation can only be run for a finite amount of time, the exact true solution can never be known. This issue gives rise to two important questions: What is the accuracy in the results of a simulation's output? How long should a simulation be run in order to obtain a given accuracy? These questions can be answered using confidence intervals and run-length control algorithms.

Using an ad hoc technique instead of the methods described in this section can be dangerous as well as wasteful. Running a simulation for too short an amount of time can result in performance statistics that are highly inaccurate. Running a simulation for an unnecessarily long amount of time wastes computing resources and delays the completion of the simulation study. And, without some type of formal analysis, the errors in simulation results cannot be quantified.

16.1. Confidence Intervals

A *confidence interval* is a range of values in which the true answer is believed to lie with a high probability. The interval can be specified in two equivalent ways, either by specifying the midpoint of the interval (which could be considered the "best guess" for the true answer) and the half-width of the interval, or by specifying the lower and upper

bounds of the interval. CSIM reports the confidence interval in both formats, as illustrated below:

```
4.114119 +/- 0.296434 = [3.817684, 4.410553]
```

The probability that the true answer lies within the interval is called the *confidence level*. Since a confidence level of 100% would result in an infinitely wide confidence interval, confidence levels from 90% to 99% are most often used. Be aware that there is always a small probability (dictated by the confidence level) that the true answer lies outside the confidence interval.

Confidence intervals can be automatically generated for the mean values in any table, qtable, meter, or box simply by calling one of the following functions immediately after the statistics object has been initialized.

Example: `m_table.confidence()`

Example: `m_qtable.confidence()`

Example: `m_meter.confidence()`

Example: `m_box.time_confidence()`

Example: `m_box.number_confidence()`

The technique used to calculate confidence intervals is called *batch means analysis*. It is beyond the scope of this manual to describe the mathematics underlying this technique, but any good simulation text should provide details.

If confidence intervals have been requested for a table, qtable, meter, or box, the statistics report will include a section like the following:

confidence intervals for the mean after 50000 observations

level	confidence interval	rel. error
90 %	4.114119 +/- 0.296434 = [3.817684, 4.410553]	0.077648
95 %	4.114119 +/- 0.354041 = [3.760078, 4.468159]	0.078837
98 %	4.114119 +/- 0.421555 = [3.692563, 4.535674]	0.080279

Notice that confidence intervals are calculated for three commonly used confidence levels: 90%, 95%, and 98%. The confidence intervals are reported in both of the formats described previously. The relative error measures the accuracy in the midpoint of the interval as an estimate of the true answer. It is defined to be the half-width divided by the lower bound of the interval. Like any relative error, its value suggests how many accurate digits there are in the estimate.

The algorithm for computing confidence intervals groups the observations into fixed size batches and uses only complete batches. For this reason, the number of observations used in the calculation of the confidence intervals may be slightly less than the number of observations used in computing the other performance statistics. For example, in the above report, 50,000 observations were used to calculate the confidence intervals. The part of the report not shown may give the mean, variance, standard deviation, etc., based on 50,472 observations.

The algorithm also requires a minimum number of observations for its results to be valid. This minimum number cannot be known before running the simulation because it depends on the amount of correlation found in the statistic. If a report is produced before sufficient observations have been obtained, the message

```
> insufficient observations to compute
        confidenceintervals
```

will appear in place of the confidence intervals. To obtain confidence intervals, run the simulation longer or use the run length control algorithm.

16.2. Inspector Functions

All values calculated by the confidence interval algorithm can be retrieved during the execution of a model or upon its completion. For a table object *m_table* and confidence level *cl*:

Exampe:	**Functional value:**
`int m_table.batch_size()`	size of batch
`int m_table.batch_count()`	number of batches
`double m_table.conf_mean()`	midpoint of interval
`double m_table.conf_halfwidth(cl)`	half-width of interval
`double table.conf_lower(cl)`	lower bound of interval
`double table.conf_upper(cl)`	upper bound of interval
`double table.conf_accuracy(cl)`	accuracy achieved

For the QTable object *m_qtable* and confidence level *cl*:

Example:	**Functional value:**
`int m_qtable.batch_size()`	size of batch
`int m_qtable.batch_count()`	number of batches
`double m_qtable.conf_mean()`	midpoint of interval
`double m_qtable.conf_halfwidth(cl)`	half-width of interval
`double m_qtable.conf_lower (cl)`	lower bound of interval
`double m_qtable.conf_upper(cl)`	upper bound of interval
`double m_qtable.conf_accuracy(cl)`	accuracy achieved

If confidence intervals have not been requested or if there have not been sufficient observations to calculate confidence intervals, all of the above functions return zero values.

To inspect confidence interval information for meters and boxes, pass to the appropriate function listed above a pointer returned by one of the following functions: *meter::ip_table, box::time_table,* or *box::number_qtable.*

16.3. Run Length Control

If the reported confidence intervals show that the needed accuracy has not been achieved, a simulation could be run again for a longer amount of time. A second, longer simulation has two disadvantages: repeating part of the simulation is wasteful, and it may not be clear how much longer to run the simulation the second time.

A better method is to use the *run length control* algorithm that is built into CSIM. This algorithm monitors the confidence interval as it narrows and automatically terminates the simulation when the desired accuracy has been achieved.

To use run length control, choose a performance measure that will be used to decide when the simulation should terminate. Instrument the model to gather statistics on this performance measure using a table, qtable, meter, or box. Immediately after the statistics gathering object has been initialized, call the appropriate function below.

For a *Table* object:

Prototype:
```
void run_length(double accuracy, double
    conf_level, double max_time)
```

Example:
```
m_tbl.run_length (0.01, 0.95, 10000.0);
```

For a *QTable* object:

Prototype:
```
void  run_length(double accuracy, double
    conf_level, double max_time)
```

For a *Meter* object:

Prototype:
```
void  run_length(double accuracy, double
    conf_level, double max_time)
```

For a *Box* object:

Prototype:
```
void  time_run_length(double accuracy,
double
    conf_level, double max_time)
```

16. Confidence Intervals and Run Length Control

Prototype: `void box::number_run_length(double, accuracy,`
`double conf_level, double, max_time)`

The accuracy parameter specifies the maximum relative error that will be allowed in the mean value of this performance measure. A value of *0.1* is usually used to request one digit of accuracy, *0.01* is used to request two digits of accuracy, and so forth. The *conf_level* parameter is the confidence level and usually has a value between *0.90* and *0.99*. The *max_time* parameter places an upper bound on how long the simulation will run. If the specified accuracy cannot be achieved within this time, the simulation will terminate and a warning message will appear in the report.

In the main CSIM process, place the following call to the *wait* function.

```
model.converged.untimed_wait();
```

Converged is a built-in event (in the *Model* object) that does not need to be declared or initialized. This event is set when the run length control algorithm determines that the requested accuracy has been achieved or when the maximum time has passed.

If run length control has been enabled, the statistics report will include a section like the following:

results of run length control using confidence intervals

cpu time limit	10.0	accuracy requested	0.005000
cpu time used	1.8	accuracy achieved	0.005000

95.0% confidence interval: 0.998735 +/- 0.004969 = [0.993767, 1.003704]

The confidence interval is reported in both formats for the confidence level that was specified. If the requested accuracy was not achieved or if there were not enough observations to calculate confidence intervals, a warning message will appear in the report.

The mechanics for running a simulation until multiple performance measures have been obtained to desired accuracy are simple. Call the appropriate run length function for several statistics gathering objects and then wait on the “converged” event as many times as there are statistics to converge. However, there are some subtleties in the theory underlying this procedure. Persons interested in this topic should read the section on output analysis in the textbook, *Simulation Modeling and Analysis,* by Law and Kelton [LaKe 99].

16.4. Caveats

Confidence intervals attempt to bound the errors in performance statistics caused by running a simulation for a finite amount of time. They in no way measure the errors caused by the model being an unfaithful representation of the actual system.

All known techniques for computing confidence intervals are heuristics. Detecting and removing correlation from performance data is a mathematically difficult problem. Confidence intervals should always be considered to be estimates.

In spite of these limitations, it is our belief that confidence intervals and run length control play an essential role in any simulation study. Simply running a simulation for a “long time” and hoping that the performance measures will be highly accurate is an unprofessional and dangerous approach.

17. Process Classes

Process classes are used to segregate data for reporting purposes. A set of usage statistics is automatically maintained for each process class. These are "printed" whenever a *report* or a *report_classes* statement is executed. In addition, facility information (from *report_facilities)* is kept by process class, when process classes exist. Obtaining usage statistics for process classes at the facilities is the most common use of this feature. See section 19.3, *CSIM Report Output*, for details about the generated reports.

17.1. Declaring and Initializing Process Classes

To declare a process class:

Example: `ProcessClass c;`

A process class must be initialized via the *ProcessClass* constructor statement before it can be used in any other statement.

Prototype: `ProcessClass(String name)`

Example:
```
ProcessClass c = new ProcessClass(
                          "low priority")
```

17.2. Using Process Classes

To have the executing process join a process class:

Prototype: `void setProcessClass(ProcessClass c)`

Example: `setProcessClass(c);`

To retrieve a pointer to the ProcessClass within an active process:

Prototype: `ProcessClass getProcessClass()`

Example: `ProcessClass c = getProcessClass();`

If no *setProcessClass* statement is executed for a process, that process is automatically a member of the "default" class. A *report* statement will not print process class statistics for the default process class. A *report_classes* statement will print process class statistics for the default process class, but ONLY if it is the only process class. If any other process class is defined, *report_classes* will only report on non-default process classes.

Note: It is necessary to use the *collect* method at a facility or the *collect_class_facility_all* statement to obtain usage by process classes at the facilities (see section 4.11, *Collecting Class-Related Statistics*).

17.3. Producing Reports

Reports for process classes are most often produced by calling the *report* function, which prints reports for all of the CSIM objects. Reports can be produced for all existing process classes by calling the *report_classes* function. The report for a process class gives the

class id, the class name, the number of entries into the class, the average lifetime for a process in this class, the average number of hold operations executed by jobs in this class, the average time per hold and the average wait time per job in this class.

PROCESS CLASS SUMMARY

id	name	number	lifetime	hold count	hold time	wait time
0	default	493	4.05680	0.99594	4.05680	0.00000
1	low priority	293	229.66986	0.54266	2.27873	227.39113
2	high priority	198	2.18412	1.00000	1.67845	0.50567

17.4. Resetting Process Classes

The statistics associated with a process class can be reset as follows:

Prototype: `void reset()`

Example: `c.reset();`

The statistics associated with all of the process classes can be reset as follows:

Prototype: `void reset_process_classes()`

Example: `reset_process_classes();`

17.5. Inspector Methods

These methods each return a statistic that describes some aspect of the usage of the specified process class. The type of the returned value for each of these methods is indicated.

Prototype:	Functional Value:
`int id()`	id of process class
`String name()`	pointer to name of process class
`int cnt()`	number of processes in process class
`double lifetime()`	total time for all processes in process class
`int holdCnt()`	total number of holds for all processes in process class
`double holdTime()`	total hold time for all processes in process class

18. Random Numbers

Most simulations are random number driven. In such simulations, random numbers are used for interarrival times, service times, allocation amounts, and routing probabilities. For each application of random numbers in a simulation, a distribution must be chosen. The distribution determines the likelihood of different values occurring. A distribution is uniquely specified by the name of its family (such as uniform, exponential, or normal) and its parameter values (such as the mean and standard deviation). Discussions of distributions and their uses in models can be found in many textbooks [LaKe 99].

Random numbers generated by computers are actually *pseudo-random*. A sequence of values is generated using a recurrence relation that calculates the next value in the sequence from the previous value. The sequence is begun by specifying a starting value called a *seed*. A good random number generator has the property that the numbers it produces have no discernible patterns that distinguish them from truly random numbers.

Most CSIM users need only read the following two sections, which describe *single stream* random number generation. Those interested in building *multiple stream* simulations should read the remaining sections as well.

18.1. Single Stream Random Number Generation

CSIM includes a library of functions for generating random numbers from more than 11 different distributions. Continuous distributions have values that are floating-point numbers; values from these distributions are most often used for amounts of time. Discrete distributions have values that are integers; values from these distributions are often used for quantities of resources.

The following prototypes are used for the functions that generate values from continuous distributions. The parameters *min* and *max* specify the minimum and maximum values that will be generated. The parameters *mean*, *var*, *stddev*, and *mode* specify respectively the mean, variance, standard deviation, and mode of the distribution. All of these distributions are methods in the *Random* class; the default object in the *Random* class is *rand*. The *rand* object has a number of methods that produce samples from different probability distributions.

Prototype: `double uniform(double min, double max)`

Example: `double x = rand.uniform(double min, double max)`

Prototype:
```
double triangular(double min, double max,
  double mode)
```

Prototype: `double exponential(double mean)`

Prototype: `double erlang(double mean, double var)`

Prototype: `double hyperexponential(double mean, double var)`

Prototype: `double normal(double mean, double stddev)`

Prototype: `double lognormal(double mean, double stddev)`

Prototype: `double hypoexponential(double mn, double var)`

The following prototype is for the function that generates values from a discrete distribution. The parameters *min* and *max* specify the minimum and maximum values that will be generated. The parameter *mean* specifies the mean of the distribution.

Prototype: `int uniform_int(int min, int max)`

Example: `int n = rand.uniform_int(int minV, int maxV)`

18.2. Changing the Seed of the Single (Default) Stream

By default, the single stream (the *rand* object) from which all random numbers are generated is seeded with the value of *1*. Unless the seed is changed, every execution of every CSIM program will use the same sequence of random numbers. The seed can be changed by calling the *setSeed* method in the *rand* object.

Prototype: `void setSeed(int n)`

Example:
```
rand.setSeed(13579);
```

The parameter is the positive integer that is to be used as the seed. The choice of the seed value will not affect the randomness of the numbers that are produced. Although it is most common to call *setSeed* once at the beginning of a CSIM program, the *setSeed* function can be called any number of times and from any place within a program.

18.3. Single versus Multiple Streams

In a *single stream* simulation, all random numbers are produced from a single stream of pseudo-random integers. The random numbers used for a particular purpose (for example, interarrival times) are generated from a subsequence of these random integers. It is of concern to some people that the subsequence of integers may not be “as random” as the stream from which they were extracted. This concern can be alleviated by using multiple separate streams of pseudo-random integers, one for each application of random numbers in the model. So, these multiple separate streams would be used for the service times at each facility, for the allocation amounts of each storage, and so forth.

Multiple streams are also used to guarantee that exactly the same sequence of random numbers is used for the interarrival times (for example) in two different models. This technique is called *common random numbers* and is described in simulation texts.

There is virtually no difference in the time required to generate random numbers from a single stream or from multiple streams. Multiple stream simulations require slightly more programming: the multiple streams must be declared, initialized, and (perhaps) seeded, and each call to a function that generates random numbers must specify the stream to be used.

18.4. Managing Multiple Streams

A stream of random numbers is declared in a CSIM program using the class *Random*.

Example: `Random *s;`

Before a stream can be used, it must be initialized by instantiating a new *Random* object:

Prototype: `Random() or Random(int s)`

Example: `Random m_s = new Random();`

By default, *Random* objects are created with seeds that are spaced 100,000 values apart. CSIM contains a table of 100 such seed values; if more than 100 streams are created, the seed values are reused.

The seed value for any stream (*Random* object) can be changed by calling the *setSeed* function.

Prototype: `void setSeed(int n)`

Example: `m_s.setSeed(24681);`

The parameter is a positive integer that is to be used as the new seed. Although it is most common to call *setSeed* once for each stream at the beginning of a CSIM program, streams can be reseeded any number of times and at any place in the program.

18.5. Multiple Stream Random Number Generation

The same distributions are available for generating random numbers from multiple streams as are available for generating random numbers from the default (single) stream.

In all ways, the methods and their parameters are exactly the same for single stream and multiple stream use. It is the programmer's responsibility to ensure that a stream is used for only one purpose and that a separate stream is used for each application of random numbers in the model, if this is the desired behavior.

19. Output from CSIM and the *Model* Class

In order for a simulation model to be useful, output indicating what occurred must be produced so that it can be analyzed. The following kinds of output can be produced from CSIM:

- Reports

CSIM always collects usage and queueing information on facilities and storage units. In addition, it will collect summary information from tables, qtables, histograms and qhistograms, if any were created by the user. All of this information can be printed via various report statements.

- Status reports

Throughout the execution of the model, CSIM collects information on current status. This information will be printed via various status statements.

If no report statement is specified, CSIM will not generate any output (although the user can generate customized output by gathering data through the various information retrieval statements, doing calculations on it if desired, and printing it).

19.1. The *Model* Class

Every CSIM model extends the *Model* class. This class provides much of the functionality required by the model. The *Model*

constructor is normally invoked by using the *super* method from the CSIM *model* constructor:

Example:

```
public class App extends Model {
    public static void main(String args[]){
        App  model = new App();

        ...
    }
    public App() {
        super("App");
    }
    public run() {
        start(new Sim()):
    }
    ...
}
```

The *start* method (in the *Model* class) is used to initiate the *first* process in the model. The other processes are initiated by the *add* method.

Prototype:

```
public class Model()
public class Model(String name)
```

Prototype:

```
public void start(Process p)
public void add(Process p)
```

The *run* method in the model is invoked by the Java thread package. Every CSIM process must have a *run* method.

The *Model* class maintains the simulation time clock; at any time, the value of this clock is the current point in simulated time. The value of this clock is available through the *clock()* method:

Prototype:

```
double clock()
```

Example:

```
double startTime = model.clock();
```

19.2. Generating Reports

19.2.1. Partial Reports

A partial report can contain information on just one type of object or just the header, using the following methods in the *Model* class:

Prototype: `void report_hdr()`

Example: `model.report_hdr()`

Prototype: `void report_facilities()`

Prototype: `void report_storages()`

Prototype: `void report_buffers()`

Prototype: `void report_classes()`

Prototype: `void report_events()`

Prototype: `void report_mailboxes()`

Prototype: `void report_tables()`

Prototype: `void report_qtables()`

Prototype: `void report_meters()`

Prototype: `void report_boxes()`

Where:

- *report_hdr()* prints the header of the report
- *report_facilities()* prints the usage statistics for all facilities defined in the model
- *report_storages()* prints the usage statistics for all storage units defined in the model

- *report_buffers()* prints the usage statistics for all buffers defined in the model
- *report_classes()* prints the process usage statistics for all process classes defined in the model
- *report_events()* prints the usage statistics for all events defined in the model and for which the monitor method has been invoked
- *report_mailboxes()* prints the usage statistics for all mailboxes defined in the model and for which the *monitor* method has been invoked
- *report_tables()* prints the summary information for all tables (with histograms and confidence intervals)
- *report_qtables()* prints the summary information for all qtables (with histograms and confidence intervals)
- *report_meters()* prints the summary information for all meters (with histograms and confidence intervals)
- *report_boxes()* prints the summary information for all boxes (with histograms and confidence intervals)

Notes:

Details of the contents of these reports can be found in section 19.3, *CSIM Report Output*.

19.2.2. Complete Reports

A complete report contains all of the sub-reports.

Prototype: `void report()`

Example: `model.report()`

Notes:

- The sub-reports appear in the order:

- *report_hdr*
- *report_facilities*
- *report_storages*
- *report_buffers*
- *report_classes*
- *report_events*
- *report_mailboxes*
- *report_tables*
- *report_qtables*
- *report_meters*
- *report_boxes*

- Details of the contents of these reports can be found in section 19.3, *CSIM Report Output*.

19.2.3. To Change the Model Name:

Prototype: `void setName(String name)`

Example: `model.setName("prototype system");`

Where:

- *name* is the new name for the simulation model (as a quoted string or type *char**)

Notes:

name appears as the model name in the report header (in *report_hdr* and *report).*

Unless changed by this statement, the model name will be the name specified by the *Model* constructor. If no name is specified for the *Model* constructor, the model name will be *def.*

To retrieve the current model, use the *name* method:

Prototype: `String name()`

Example: `String m_name = model.name();`

19.3. CSIM Report Output

The output generated by the report statements present information on the simulation run as it has progressed so far. The sub-reports that comprise the overall report are:

- Header
- Report on facility usage (if any facilities were declared)
- Report on storage usage (if any storage units were declared)
- Report on buffer usage (if any buffers were declared)
- Report on the process classes (if more than one process class (the default process class) has been declared)
- Report on the monitored events
- Report on the monitored mailboxes
- Summary for each table (with histogram and confidence interval) declared
- Summary for each qtable (with histogram and confidence interval) declared
- Summary for each meter (with histogram and confidence interval) declared
- Summary for each box (with histogram and confidence interval) declared

The following tables give a complete description of each of these sub-reports.

19.3.1. Report_Hdr Output

Output Heading	Meaning
Revision	CSIM version number
System	System the simulation was run on, *e.g.*, SUN Sparc
Model	Model name (see *Model* constructor or *set_model_name* statement)
Date and time	Date and time that report was printed
Ending Simulation Time	Total simulated time
Elapsed Simulation Time	Simulated time since last reset
CPU Time	Real CPU time used since last report

19.3.2. Report_Facilities Output

Output Heading	Meaning
Facility Summary	
facility name	Name (for a facility set, the index is appended)
service discipline	Service discipline (when one was defined)
Means	
service time	Mean service time per request
util	Mean utilization (busy time divided by elapsed time)
throughput	Mean throughput rate (completions per unit time)
queue length	Mean number of requests waiting or in service
response time	Mean time at facility (both waiting and in service)
Counts	
completion count	Number of requests completed

Notes:

- When computing averages based on the number of requests for facilities, the number of completed requests is used. Thus, any requests that are waiting or in progress when the report is printed do not contribute to these statistics.
- If collection of process class statistics is specified, then the above items are repeated on a separate line for each process class that uses the facility.

19.3.3. Report_Storages Output

Output Heading	Meaning
Storage Summary	
storage name	Name of storage unit
size	Size of storage unit
Means (see note below)	
alloc amount	Mean amount of storage per *allocate* request
alloc count	Number of *allocates*
dealloc amount	Mean amount of storage per *deallocate* request
util	Average percentage of storage allocated over time
in-que length	Mean time requests are waiting
in-queue time	Average time a request waits

Notes:

- When computing averages based on the number of requests for storage, the number of completed requests is used. Thus, any requests that are waiting or in progress when the report is printed do not contribute to these statistics.

19.3.4. Report_Buffers Output

Output Heading	Meaning
Buffer Summary	
buffer name	Name of storage unit
size	Size of storage unit
Means	
get amt	Mean amount of space per *get*
get qlen	Average queue length for *gets*
get resp	Average response time for *gets*
get count	Number of *gets*
put amt	Mean amount of space per *put*
put qlen	Average queue length for *puts*
put resp	Average response time for *puts*
put count	Number of *puts*

19.3.5. Report_Classes Output

Output Heading	Meaning
Process class summary	
id	Process class id
name	Process class name
number	Number of processes belonging to this class
lifetime	Mean simulated time per process in this class
hold ct	Mean number of *hold* statements per process in this class
hold time	Mean hold time per process in this class
wait time	Mean wait time per process in this class (*lifetime - holdtime*)

Notes:

- If no process classes are specified, the report for the *default* class (every process begins as a member of this class) is not provided. If any process classes are specified, then the report includes the default class.

19.3.6. Report_Events Output

Output Heading	Meaning
Event Summary	
Event name	Name of event
Number of queue vst	Total number of entries to *queue* queue
Avg queue length	Average length of *queue* queue
Avg time queued	Average time in *queue* queue
Number of wait vsts	Total number of entries to *wait* queue
Avg wait length	Average length of *wait* queue
Avg time waiting	Average time in *wait* queue
Number of set ops	Number of set operations at event

19.3.7. Report_Mailboxes Output

Output Heading	Meaning
Mailbox Summary	
Mailbox name	Name of mailbox
Number of proc visits	Number of processes doing receives
Process qlength	Average process queue length
Process rspTime	Average process response time
Number of messages	Number of messages sent
Message qlength	Average number of messages in the mailbox
Message rspTime	Average time for messages in the mailbox

19.3.8. Report_Tables Output

Output Heading	Meaning
Tables	
minimum	Minimum value recorded
maximum	Maximum value recorded
range	*Maximum - minimum*
observations	Number of entries in table
mean	Average of values recorded
variance	Variance of values recorded
standard deviation	Square root of variance
coefficient of var.	Standard deviation divided by the mean
Confidence Intervals	
Observations	Number of observations used to compute interval
Level	Probability that interval contains true mean
Confidence interval	Two forms: *Mid-point +/- half-width* *Lower limit - upper limit*
Re. error	Relative error: *half-width divided by lower limit*

Histograms	
Lower limit	Low value for this bucket
Frequency	Number of entries in this bucket
Proportion	Fraction of total number of entries that are in this bucket
Cumulative proportion	Fraction of total number of entries that are in this bucket and all lower buckets

19.3.9. Report_Qtables Output

Output Heading	**Meaning**
Qtables and Qhistograms	
Initial	Initial state value
Final	Final state value
Entries	Number of entries to states
Exits	Number of exits from states
Minimum	Minimum state value
Maximum	Maximum state value
Range	Range of state values
Mean	Mean state value (time-weighted)
Variance	Variance of state values
Standard deviation	Square root of variance
Coeff. of variation	Coefficient of variation: *standard deviation divided by mean*

19. Output from CSIM

Confidence Intervals	
Observations	Number of observations used to compute interval
Level	Probability that interval contains true mean
Confidence Interval	Two forms: *Mid-point +/- half-width* *Lower limit - upper limit*
Rel. error	Relative error: half-width divided by lower limit
Histograms	
Lower limit	Low value for this bucket
Frequency	Number of entries in this bucket
Proportion	Fraction of total number of entries that are in this bucket
Cumulative proportion	Fraction of total number of entries that are in this bucket and all lower buckets

Notes:

- All histogram output for qtables is grouped by state value, where each interval except the last includes only one state value. The last bucket contains all state values greater than the value covered by the penultimate value.

19.3.10. Report_Meters Output

Output Heading	Meaning
Meters	
Count	Number of entries noted
Rate	Rate at which entries are noted
Interpassage time statistics	(see *Tables* table above)
Confidence Intervals	(see *Tables* table above)
Histograms	(see *Tables* table above)

19.3.11. Report_Boxes Output

Output Heading	Meaning
Boxes	
Statistics on elapsed times (see *Tables* table above)	
Confidence Intervals	(see *Tables* table above)
Histograms	(see *Tables* table above)
Statistics on population (see Qtables)	
Confidence Intervals	(see *Qtables* table above)
Histograms	(see *Qtables* table above)

19.4. Redirecting Output Files

By default, CSIM output is written to file *stdout*. The CSIM output can be redirected to a different file using the method *setOutputFile*.

Prototype: `void setOutputFile(String fileName)`

Example:
```
model.setOutputFile("model.out");
```

The output can also be directed to a print stream:

Prototype: `void setOutputStream(PrintStream s)`

Example:
```
PrintStream s = Files.Setfile("model.out");
model.setOutputStream(s);
```

The *stream* form is useful if the model will be generating output in addition to the standard CSIM output.

19.5. Generating Status Reports

The *status_reports* methods in the *Model* class can be used to give details on the state of the objects in the model.

19.5.1. Partial Reports

Prototype: `void status_next_event_list()`

Example: `model.status_next_event_list()`

Prototype: `void status_events()`

Prototype: `void status_mailboxes()`

Prototype: `void status_facilities()`

Prototype: `void status_storages()`

Prototype: `void status_buffers()`

Where:

- *status_next_event_list* prints the pending state changes for processes
- *status_events* prints the status of all events defined in the model
- *status_mailboxes* prints the status for all mailboxes defined in the model
- *status_facilities* prints the status of all facilities defined in the model
- *status_storages* prints the status of all storages defined in the model
- Details of the contents of these reports are in the sections of this document that discuss their related objects.

19.5.2. Complete Reports

Prototype: `void dump_status()`

Notes:

- The sub-reports appear in the order:
  - *status_next_event_list*
  - *status_events*
  - *status_mailboxes*
  - *status_facilities*
  - *status_storages*
  - *status_buffers*

Each of the above status statements is callable, so a "customized" status report can be created.

20. Tracing Simulation Execution

A simulation program, like any other complex software, can be difficult to debug and verify that it is correct. To aid in this, CSIM can produce a log of trace messages during the execution of a simulation. A one-line trace message is produced each time an interesting change in the state of the simulation occurs.

Even a short simulation run can generate an enormous number of trace messages. For this reason, you should try to be selective when enabling different tracing options.

20.1. Tracing all State Changes

The generation of trace messages for all state changes is enabled using the *trace_on* function. The tracing is disabled using the *trace_off* function.

Prototype: `void enableTrace(boolean attr)`

Example: `model.enableTrace(true);`

Example: `model.enableTrace(false);`

Trace messages can be turned on and off as desired during a simulation. Logic can even be added to a simulation to turn on trace messages when a specific condition is detected.

20.2. Format of Trace Messages

Each trace message contains the current simulation time, the name and sequence number (id) and the priority of the process that caused the state change, and a description of the state change. Sample trace messages are shown below.

```
time      process    id   pri   status

0.716     customer   4    1     use facility cpu for 0.070

0.716     customer   4    1     reserve facility cpu

0.716     customer   4    1     hold for 0.070

0.716     customer   4    1     sched proc: t = 0.070, id = 4

0.787     customer   4    1     release facility cpu
```

20.3. What Is and Is Not Traced

Ideally, every occurrence that changes the state of a CSIM object will generate a trace message. In particular, any occurrence that causes time to pass should be traced.

Occurrences that do not produce trace messages include 1) the generation of random numbers, 2) the updating of performance statistics, and 3) the production of reports. Obviously, non-CSIM operations such as updates of local variables cannot produce trace messages.

20.4. Redirecting Trace Output

By default, trace messages are written to file *stdout*. Trace messages can be redirected to a different file using the methods *setTraceFile* and *setTraceStream*.

Prototype: `void setTraceFile(String fileName)`

Example:
```
model.setTraceFile("trace.out");
```

The trace messages can also be directed to a print stream:

Prototype: `void setTraceStream(PrintStream s)`

Example:
```
PrintStream s = Files.Setfile("trace.out");
model.setTraceStream(s);
```

20. Tracing Simulation Execution

21. MISCELLANEOUS

21.1. Real Time

Although internally the model only deals with simulated time, the running of the model takes place in real time.

21.1.1. To Retrieve the Current Real Time:

Prototype: `String time_of_day()`

Example: `String tod = model.timeOfDay();`

Where:

- *tod* is the actual time of day string

Notes:

- The format of the returned string is:

  `Month dd,yyyy hh:mm:ss AM/PM Timezone`

  Example:
  `August 14, 2005 9:32:46 PM CDT.`

21.1.2. To Retrieve the Amount of CPU Time Used by the Model:

Prototype: `double executionTime()`

Example: `double t = model.executionTime();`

Where:

- *t* is the amount of CPU time, in seconds, that has been consumed by the model thus far (type *double*)

21.2. Creating a CSIM Program

The usual way to create a CSIM for Java program is as follows:

- Create a Java program file (*Name*.java)
- In the file, insert the following import statements:
  - `import com.mesquite.csim.*;`
  - `import com.mesquite.csim.Process;`
  - `import com.mesquite.csim.file.Files;`
  - `import java.io.*;`
- In the file, create the public class *Name* as follows:
  - ```
    public class Name extends Model {
        public static void main(String args[]) {
            model = new Name();
            model.run();
            model.report();
        }
        public Name() {
            super("Name");
    ```
    ```

```
    }
    public void run() {
        start(new Sim());
    }
    private static Name model;
    ...
```

- Add a *Sim* process to the model:
  - ```
    private class Sim extends Process {
        public Sim() {
            super("Sim"):
        }
        public void run() {
            // code in Sim process
        }
    ```
  - The tasks of the *Sim* process could include the following:
    - Instantiate the simulation objects, such as facilities, etc.
    - Add additional processes to the model
    - Allow the model to operate until the ending condition occurs
    - Exit (back to the *run* method in the *Name* object)

21.3. Rerunning or Resetting a CSIM Model

It may be useful to run a model multiple times with different values, or run multiple models in the same program.
    ```

21.3.1. To Rerun a CSIM Model:

The model can be initiated as many times as necessary. Every time the *model.run()* statement is executed, the model is reinitialized.

21.3.2. To Clear Statistics without Rerunning the Model:

Prototype: `void reset()`

Example: `model.reset();`

Notes:

- *reset* will cause the following to occur:
  - All statistics for facilities and storage units are cleared.
  - All non-permanent table structures are cleared
  - All remaining facilities, storage units, events, etc., are eliminated
  - The simulated time clock is set to zero
- The variable *clock* is not altered.
- Time intervals for facilities, storage units and qtables that began before the *reset* are tabulated in their entirety if they end after the *reset*.
- This feature can be used to eliminate the effects of start-up transients.

21.4. Error Handling

Currently, all CSIM errors are detected and execution is stopped. It is anticipated that in a future version of CSIM for Java, the programmer will have the option of "catching" CSIM errors.

21.5. Output File Selection

CSIM allows the user to select where various types of output should be sent. The default file for all of these is *stdout*. The following files can be specified:

- *Output file* - for reports and status dumps
- *Error file* - for error messages
- *Trace file* - for traces

21.5.1. To Change the Stream to which a Given Type of Output is Sent:

Prototype: `void setOutputStream(PrintStream ps)`

Prototype: `void setTraceStream(PrintStream ps)`

Example:
```
PrintStream ps = Files.Setfile("file.txt");
model.setOutputStream(ps);
model.setTraceStream(ps);
```

Where:

- *ps* is an instance of PrintStream

21.6. Running Java for CSIM Programs

A CSIM program needs access to the CSIM package in order to both compile a model into class files and to run it. A program is normally compiled with the Java archive (jar) file named `csimForJava.jar`. If the command line technique is used to compile and run the model, the command would be:

```
javac -classpath <path for csimForJava.jar>;.
   prog.java
```

The program can be executed using the command:

```
java -classpath <path for csimForJava.jar>;. prog
```

22. Error Messages

A CSIM program that detects a problem can print the following error messages. Currently, a Runtime exception is thrown. In future versions, there will be provisions for the program to catch and handle these errors. Right now, the program prints the message associated with the error and then terminates.

22.1. Runtime Exceptions

The following runtime exceptions can be thrown by CSIM for Java:

- currentModel: Not inside any model
- checkThread: current thread not active thread
- schedule: interval cannot be negative
- facility: tried to free unassigned server
- hyperexponential: var < mean*mean
- triangular: parameter error
- uniform: max < min
- uniform_int: max < min
- storage: allocate amt > capacity
- storage: deallocate amt + available > capacity

22.2. Illegal State Exceptions

The following illegal state exceptions can be thrown by CSIM for Java:

- add: model not running
- start: model already running
- setOutputStream: not initialize
- nextEvent: No more events
- eventSet; could not find event

22.3. Print Message and Exit

CSIM for Java can generate the following message:

- setTraceFile: could not open trace file

23. Acknowledgments

- Teemu Kerola assisted in the initial implementation of CSIM. He also designed and implemented the MONIT event logging feature and the post-run analysis program for the SUN.
- Bill Alexander has provided consultation on the wisdom of many proposed features.
- Leonard Cohn suggested using mailboxes.
- Ed Rafalko of Eastman Kodak provided the changes required to have CSIM available on the VMS operating system.
- Rich Lary and Harry Siegler of DEC have provided code for the VMS version of CSIM. They also suggested a number of modifications that have improved the performance of CSIM programs.
- Geoff Brown of Cornell University did most of the work for the HP-300 version. He also provided the note on CSIM on the NeXT System.
- Jeff Brumfield of The University of Texas at Austin critiqued many aspects CSIM. He and Kerola suggested process classes.
- Connie Smith of L & S Systems did much of the work on the Macintosh version.
- Kevin Wilkinson of HP Labs did most of the work on the HP Prism support.
- Murthy Devarakonda of IBM T.J. Watson Research Labs did most of the work on IBM RS/6000 support.
- Jeff Brumfield provided the ideas, code, and documentation on meters, boxes, confidence intervals, and run length control. He

23. Acknowledgments

also improved the format of the output reports and added the additional probability distributions.

- Beth Tobias rewrote the CSIM manual.
- Jorge Gonzales helped test and debug CSIM18.
- Dawn Childress revised and reformatted the CSIM18 manuals.
- Conor Davis led the design and implementation of CSIM for Java
- Lisa Wells edited and helped produce the CSIM for Java documentation.

24. List of References

[Brow88] Brown, R., Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem, *Communications of the ACM*, (31, 10), October, 1988, pp. 1220-1227.

[KeSc87] Kerola, T. and H. Schwetman, Monit: A Performance Monitoring Tool for Parallel and Pseudo-Parallel Programs, *Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, ACM/SIGMETRICS, May, 1987, pp. 163-174.

[LaKe99] Law, A. and D. Kelton, *Simulation Modeling and Analysis,* third edition, (McGraw-Hill, 1999).

[MaMc73] MacDougall, M.H. and J.S. McAlpine, Computer System Simulation with ASPOL, *Symposium on the Simulation of Computer Systems*, ACM/SIGSIM, June, 1973, pp. 93-103.

[MacD74] MacDougall, M.H., Simulating the NASA Mass Data Storage Facility, *Symposium on the Simulation of Computer Systems*, ACM/SIGSIM, June 1974, pp. 33-43.

[MacD75] MacDougall, M.H., Process and Event Control in ASPOL, *Symposium on the Simulation of Computer Systems,* ACM/SIGSIM, August, 1975, pp. 39-51.

24. List of References

[Schw86] Schwetman, H.D., CSIM: A C-Based, Process-Oriented Simulation Language, *Proceedings of the 1986 Winter Simulation Conference,* December, 1986, pp. 387-396.

[Schw88] Schwetman, H.D., Using CSIM to Model Complex Systems, *Proceedings of the 1988 Winter Simulation Conference,* December, 1988, pp. 246-253; also available as Microelectronics and Computer Technology Corporation, Technical Report ACA-ST-154-88.

[Schw90b] Schwetman, H.D., Introduction to Process-Oriented Simulation and CSIM, *Proceedings of the 1990 Winter Simulation Conference*, December, 1990, pp. 154-157.

[Schw94] Schwetman, H.D., CSIM17: A Simulation Model-Building Toolkit, Proceedings of the 1994 Winter Simulation Conference, December, 1994. pp. 464-470.

[Schw95] Schwetman, H.D., Object-Oriented Simulation Modeling with C++/CSIM17, *Proceedings of the 1995 Winter Simulation Conference*, December, 1995.

[Schw96] Schwetman, H.D., CSIM18 - The Simulation Engine, *Proceedings of the 1996 Winter Simulation Conference*, December, 1996.

[Schw97] Schwetman, H.D., Data Analysis and Automatic Run Length Control in CSIM18, *Proceedings of the 1997 Winter Simulation Conference*, Atlanta, GA, December, 1997

24. List of References

[Schw98] Schwetman, H.D., Model-Based Systems Analysis Using CSIM18, *Proceedings of the 1998 Winter Simulation Conference*, Washington, DC, December. 1998

[Schw99] Schwetman, H.D., Model, Then Build: A Modern Approach to Systems Development Using CSIM18, *Proceedings of the 1999 Winter Simulation Conference*, Phoenix, AZ, December, 1999

[Schw00a] Schwetman, H.D., Finding the Best System Configuration: An Application of Simulation and Optimization, *Proceedings of the 2000 European Simulation Multiconference*, Gent, Belgium, May, 2000 (Society for Computer Simulation)

[Schw00b] Schwetman, H.D., Optimizing Simulations with CSIM18/OptQuest: Finding the Best Configuration, *Proceedings 2000 Winter Simulation Conference*, Orlando, FL, December, 2000

[Schw01] Schwetman, H.D., CSIM19: A Powerful Tool For Building Systems Models, *Proceedings 2001 Winter Simulation Conference*, Washington, DC, December, 2001

24. List of References

25. Sample Program

A sample CSIM for Java program follows. This program is a model of an M/M/1 queueing system. The process *gen* includes a *while* loop, which generates arriving customers at appropriate intervals (exponentially distributed with mean *iarTime*), each represented by a job process. These customers contend for the facility on a first come, first served basis. As each customer gains exclusive use of the facility, they delay for a service period (again exponentially distributed, but with mean *srvTime*) and then depart. Model behavior statistics are automatically collected at the *FCFS_facility*.

Sample Program to Simulate Single Server Facility

```
// Generic application: App.java

import com.mesquite.csim.*;
import com.mesquite.csim.Process;
import com.mesquite.csim.file.Files;
import java.io.*;

public class App extends Model {
   public static void main(String args[]) {
      App model = new App();
      m_s = Files.Setfile("App.out");
      model.setOutputStream(m_s);
      model.run();
      model.report();
   }

   public App() {
      super("App");
   }
   public void run() {
```

25. Sample Program

```
                        try {
                           start(new Sim());
                        }
                        catch (Exception e) {
                           m_s.println("csim error: " +
e.getMessage());
                        }
                     }

                     private static final double simTime = 10000.0;
                     private static final double iarTime = 2.0;
                     private static final double srvTime = 1.0;
                     private FCFSFacility m_fac;
                     private static PrintStream m_s;

                     private class Sim extends Process {
                        public Sim() {
                           super("Sim");
                        }
                        public void run() {
                           m_fac = new FCFSFacility("fac", 1);
                           add(new Gen());
                           hold(simTime);
                        }
                     }

                     private class Gen extends Process {
                        public Gen() {
                           super("Gen");
                        }
                        public void run() {
                           while(true) {
                              add(new Job());
                              hold(rand.exponential(iarTime));
                           }
                        }
                     }

                     private class Job extends Process {
```

```
      public Job() {
         super("Job");
      }
      public void run() {
         m_fac.use(rand.exponential(srvTime));
      }
   }
}
```

The output from an execution of this model is a follows:

CSIM for Java Simulation Report

App

September 20, 2005 8:19:48 AM CDT

Ending Simulation time: 10002.227
Elapsed Simulation time: 10002.227
Execution (CPU) time: 0.861

FACILITY SUMMARY

facility name	service disc	service time	util.	through-put	queue length	response time	compl count
fac	fcfs	1.00954	0.512	0.50679	1.01980	2.01229	5069

25. Sample Program

26. Reserved Words, Structures, and More

26.1. Statement and Reserved Words

Usage	Object	Section
Box(char* name)	Box	14.1
Box::enter ();	Box	14.2
Box::exit (double enterTime);	Box	14.2
Box::name();	Box	14.6
Box::number_confidence();	Box	14.5
Box::number_histogram(long nbkt, long min, long max);	Box	14.4
Box::number_moving_window(long);	Box	14.6
Box::number_Qtable();	Box	14.6
Box::number_run_length(double acc, double confLev, double maxTime);	Box	14.5
Box::report();	Box	14.3
Box::reset();	Box	14.7
Box::time_confidence();	Box	14.5
Box::time_histogram(long nbkt, double xmin, double xmax);	Box	14.4

double xmax);		
Box::time_run_length(double acc, double confLev, double maxTime);	Box	14.5
Box::time_table();	Box	14.6
Buffer(char *name, long size)	Buffer	6.1
Buffer::current();	Buffer	6.7
Buffer::get(const long amt);	Buffer	6.3
Buffer::get_count(BUFFER b);	Buffer	6.7
Buffer::get_current_count();	Buffer	6.7
Buffer::get_first_Process();	Buffer	9.4
Buffer::get_insert_Process(Process_t p);	Buffer	9.4
Buffer::get_last_Process();	Buffer	9.4
Buffer::get_remove_Process(Process_t p);	Buffer	9.4
Buffer::get_timeQueue();	Buffer	6.7
Buffer::get_total();	Buffer	6.7
Buffer::name();	Buffer	6.7
Buffer::put(const long amt);	Buffer	6.2
Buffer::put_count();	Buffer	6.7
Buffer::put_current_count();	Buffer	6.7
Buffer::put_first_Process();	Buffer	9.4
Buffer::put_insert_Process(Process_t p);	Buffer	9.4
Buffer::put_last_Process();	Buffer	9.4
Buffer::put_remove_Process(Process_t p);	Buffer	9.4
Buffer::put_total();	Buffer	6.7
Buffer::reset();	Buffer	6.5

Buffer::size();	Buffer	6.7
Buffer::timed_get(const long amt, const double t);	Buffer	6.6
Buffer::timed_put(const long amt, const double t);	Buffer	6.6
Buffer_put_timeQueue(BUFFER b);	Buffer	6.7
clock()	Utility	2.2
collect_class_Facility_all();	Facility	4.11
cputime();	Utility	21.1
dump_status();	Utility	19.5
erlang(mean,var)	Random	18.1
Event(name)	Event	7.1
Event::clear()	Event	7.7
Event::first_queue_Process();	Event	9.5
Event::first_wait_Process();	Event	9.5
Event::insert_queue_Process(Process_t p);	Event	9.5
Event::insert_wait_Process(Process_t p);	Event	9.5
Event::last_queue_Process();	Event	9.5
Event::last_wait_Process();	Event	9.5
Event::monitor();	Event	7.8
Event::name();	Event	7.11
Event::qlen();	Event	7.11
Event::queue();	Event	7.4
Event::queue_cnt();	Event	7.11
Event::queue_count();	Event	7.11
Event::queue_delay_count();	Event	7.11

Event::queue_length();	Event	7.11
Event::queue_sum();	Event	7.11
Event::queue_time();	Event	7.11
Event::remove_queue_Process(Process_t p);	Event	9.5
Event::remove_wait_Process(Process_t p);	Event	9.5
Event::reset();	Event	7.9
Event::set()	Event	7.6
Event::set();	Event	7.6
Event::state()	Event	7.11
Event::state();	Event	7.11
Event::timed_queue(double);	Event	7.5
Event::timed_queue_any(double);	Event	7.10
Event::timed_wait(double);	Event	7.3
Event::untimed_queue()	Event	7.4
Event::untimed_wait()	Event	7.2
Event::untimed_wait();	Event	7.2
Event::wait_cnt();	Event	7.11
Event::wait_count();	Event	7.11
Event::wait_delay_count();	Event	7.11
Event::wait_length();	Event	7.11
Event::wait_sum();	Event	7.11
Event::wait_time();	Event	7.11
Event_set(char* name, long numEvents)	Event_set	7.10
Event_set::count();	Event_set	7.11
Event_set::monitor();	Event_set	7.10

Event_set::name();	Event_set	7.11
Event_set::num_Events(EVENT*);	Event_set	7.11
Event_set::queue_any();	Event_set	7.11
Event_set::timed_wait_any(double);	Event_set	7.10
Event_set::wait_any();	Event_set	7.10
exponential(double mean)	Random	18.1
Facility(String name)	Facility	4.1
Facility::class_completions(Process_class*);	Facility	4.12
Facility::class_qlen(PROCESSCLASS);	Facility	4.12
Facility::class_qlength(ProcessClass);	Facility	4.12
Facility::class_resp(ProcessClass);	Facility	4.12
Facility::class_serv(ProcessClass);	Facility	4.12
Facility::class_tput(ProcessClass);	Facility	4.12
Facility::class_util(ProcessClass);	Facility	4.12
Facility::collect_class_Facility();	Facility	4.11
Facility::completions();	Facility	4.12
Facility::first_Process();	Facility	9.2
Facility::insert_Process(Process_t p);	Facility	9.2
Facility::last_Process();	Facility	9.2
Facility::name();	Facility	4.12
Facility::num_busy();	Facility	4.12
Facility::num_servers();	Facility	4.12
Facility::qlen();	Facility	4.12
Facility::qlength();	Facility	4.12
Facility::release();	Facility	4.3

Facility::release(long srvrIndex);	Facility	4.7
Facility::remove_Process(Process_t p);	Facility	9.2
Facility::reserve();	Facility	4.3
Facility::reset();	Facility	4.5
Facility::resp();	Facility	4.12
Facility::serv();	Facility	4.12
Facility::server_completions(long srvrIdx);	Facility	4.12
Facility::server_serv(long srvrIdx);	Facility	4.12
Facility::server_tput(long srvrIdx);	Facility	4.12
Facility::server_util(long srvrIdx);	Facility	4.12
Facility::service_disp();	Facility	4.12
Facility::status();	Facility	4.12
Facility::timed_reserve(double);	Facility	4.9
Facility::timeslice();	Facility	4.10
Facility::tput();	Facility	4.12
Facility::use(double t);	Facility	4.2
Facility::util();	Facility	4.12
getProcessClass();	ProcessClass	17.2
hold(double holdTime);	Process	2.3
hyperexponentail(double mean, double var)	Random	18.1
hypoexponential(double mn, double var)	Random	18.1
identity()	Process	3.7
lognormal(double mean, double stddev)	Random	18.1
Mailbox(char* name)	Mailbox	8.1
Mailbox::first_msg();	Mailbox	9.6

Mailbox::first_Process();	Mailbox	9.6
Mailbox::insert_msg(message_t m);	Mailbox	9.6
Mailbox::insert_Process(Process_t p);	Mailbox	9.6
Mailbox::last_msg();	Mailbox	9.6
Mailbox::last_Process();	Mailbox	9.6
Mailbox::monitor();	Mailbox	8.5
Mailbox::msg_cnt();	Mailbox	8.7
Mailbox::msg_count();	Mailbox	8.7
Mailbox::msg_delay_count();	Mailbox	8.7
Mailbox::msg_length();	Mailbox	8.7
Mailbox::msg_sum();	Mailbox	8.7
Mailbox::msg_time();	Mailbox	8.7
Mailbox::proc_count();	Mailbox	8.7
Mailbox::proc_delay_count();	Mailbox	8.7
Mailbox::proc_length();	Mailbox	8.7
Mailbox::proc_sum();	Mailbox	8.7
Mailbox::proc_time();	Mailbox	8.7
Mailbox::queue_cnt();	Mailbox	8.7
Mailbox::receive(long msg)	Mailbox	8.3
Mailbox::receive(long* msg);	Mailbox	8.3
Mailbox::remove_msg(message_t msg);	Mailbox	9.6
Mailbox::remove_Process(Process_t p);	Mailbox	9.6
Mailbox::reset();	Mailbox	8.6
Mailbox::send(long msg);	Mailbox	8.2
Mailbox::send(ojbect* msg)	Mailbox	8.2

26. Reserved Words

Mailbox::timed_receive(long*, double);	Mailbox	8.4
Mailbox_::name();	Mailbox	8.7
message::get_next();	Message	9.6
Meter(char* name)	Meter	13.1
Meter::cnt();	Meter	13.6
Meter::confidence();	Meter	13.5
Meter::histogram(long nbkts, double min, double max);	Meter	13.4
Meter::ip_table();	Meter	13.6
Meter::name();	Meter	13.6
Meter::note_passage();	Meter	13.2
Meter::rate();	Meter	13.6
Meter::report();	Meter	13.3
Meter::reset();	Meter	13.7
Meter::run_length(double acc, double conLev, double maxTime);	Meter	13.5
Meter::start_time();	Meter	13.6
Model.status_Buffers();	Buffer	6.8
Model.status_Events();	Event	7.12
Model.status_facilities()	Facility	4.13
Model.status_facilities();	Facility	4.13
Model.status_Mailboxes();	Mailbox	8.8
model::reset();	Utility	21.3
Model::setName(const char*);	Model	19.2.3
priority();	Process	3.6
Process::get_struct();	Process	9.1

Process::identity();	Process	9.1
Process::name();	Process	3.7
Process::name();	Process	9.1
Process::priority();	Process	9.1
Process::restart();	Process	9.1
Process::set_priority(long pr);	Process	9.1
Process::set_struct(void* strct);	Process	9.1
Process_class::cnt();	Process_class	17.5
ProcessClass(name)	ProcessClass	17.1
ProcessClass::holdcnt();	ProcessClass	17.5
ProcessClass::holdtime();	ProcessClass	17.5
ProcessClass::id();	ProcessClass	17.5
ProcessClass::lifetime();	ProcessClass	17.5
ProcessClass::name();	ProcessClass	17.5
ProcessClass::reset();	ProcessClass	17.4
Qtable(name)	Qtable	12.1
Qtable::cnt();	Qtable	12.6
Qtable::conf_accuracy(double confLev);	Qtable	12.6
Qtable::conf_halfwidth(double confLev);	Qtable	12.6
Qtable::conf_lower(double confLev);	Qtable	12.6
Qtable::conf_mean();	Qtable	12.6
Qtable::conf_upper(double confLev);	Qtable	12.6
Qtable::confidence();	Qtable	12.5
Qtable::converged();	Qtable	12.5
Qtable::cur();	Qtable	12.6

26. Reserved Words

Qtable::current();	Qtable	12.6
Qtable::cv();	Qtable	12.6
Qtable::entries();	Qtable	12.6
Qtable::exits();	Qtable	12.6
Qtable::histogram(long nbkts, long min, long max);	Qtable	12.4
Qtable::histogram_bucket(long i);	Qtable	12.6
Qtable::histogram_high();	Qtable	12.6
Qtable::histogram_low();	Qtable	12.6
Qtable::histogram_num();	Qtable	12.6
Qtable::histogram_total(Q);	Qtable	12.6
Qtable::histogram_width();	Qtable	12.6
Qtable::initial();	Qtable	12.6
Qtable::max(Q);	Qtable	12.6
Qtable::mean();	Qtable	12.6
Qtable::min();	Qtable	12.6
Qtable::name();	Qtable	12.6
Qtable::note_entry();	Qtable	12.2
Qtable::note_exit();	Qtable	12.2
Qtable::note_state(long st);	Qtable	12.2
Qtable::note_value(long v);	Qtable	12.2
Qtable::range();	Qtable	12.6
Qtable::report ();	Qtable	12.3
Qtable::reset();	Qtable	12.7
Qtable::run_length(double accur, double confidLev, double maxTime);	Qtable	12.5

Qtable::state();	Qtable	12.6
Qtable::stddev();	Qtable	12.6
Qtable::sum();	Qtable	12.6
Qtable::sum_square();	Qtable	12.6
Qtable::var();	Qtable	12.6
Random()	Random	18.4
Random::erlang(double, double);	Random	18.5
Random::exponential(double mn)	Random	18.5
Random::hypoexponential(double, double);	Random	18.5
Random::lognormal(double, double);	Random	18.5
Random::reseed(long);	Random	18.2
Random_normal(NIL,mean,stddev); **?	Random	18.5
report();	Utility	19.2
report_hdr();	Utility	19.2
reset()	Utility	21.3
set_priority(long pr);	Process	3.6
setOutputFile(FILE*);	Utility	21.5
setProcessClass();	ProcessClass	17.2
status_next_Event_list();	Utility	19.5
Storage(char *name, long amt)	Storage	5.1
Storage::add_store(long amount);	Storage	5.9
Storage::alloc(long amount);	Storage	5.2
Storage::allocate(long amount);	Storage	5.2
Storage::available();	Storage	5.8
Storage::busy_amt();	Storage	5.8

Storage::capacity();	Storage	5.8
Storage::dealloc(long);	Storage	5.3
Storage::deallocate(long);	Storage	5.3
Storage::first_Process();	Storage	9.3
Storage::insert_Process(Process_t p);	Storage	9.3
Storage::last_Process();	Storage	9.3
Storage::name();	Storage	5.8
Storage::number_amt();	Storage	5.8
Storage::qlength();	Storage	5.8
Storage::queue_cnt();	Storage	5.8
Storage::release_cnt();	Storage	5.8
Storage::release_total();	Storage	5.8
Storage::remove_Process(Process_t p);	Storage	9.3
Storage::request_amt();	Storage	5.8
Storage::request_cnt();	Storage	5.8
Storage::request_total();	Storage	5.8
Storage::reset();	Storage	5.5
Storage::time();	Storage	5.8
Storage::timed_allocate(long, double);	Storage	5.7
Storage::waiting_amt();	Storage	5.8
Table(name)	Table	11.1
Table::add_histogram(long, double, double);	Table	11.4
Table::batch_count();	Table	11.6
Table::batch_size();	Table	11.6
Table::cnt();	Table	11.6

Table::conf_accuracy(double confLev);	Table	11.6
Table::conf_halfwidth(double confLev);	Table	11.6
Table::conf_lower(double confLev);	Table	11.6
Table::conf_mean();	Table	11.6
Table::conf_upper(double);	Table	11.6
Table::confidence();	Table	11.5
Table::converged();	Table	11.6
Table::cv();	Table	11.6
Table::histogram_bucket(long);	Table	11.6
Table::histogram_high();	Table	11.6
Table::histogram_low();	Table	11.6
Table::histogram_num();	Table	11.6
Table::histogram_total();	Table	11.6
Table::histogram_width();	Table	11.6
Table::max();	Table	11.6
Table::mean();	Table	11.6
Table::min();	Table	11.6
Table::name();	Table	11.6
Table::range();	Table	11.6
table::record(double val);	Table	11.2
table::report()	Table	11.3
table::reset();	Table	11.7
Table::run_length(double, double, double);	Table	11.5
table::setPermanent	Table	11.1
Table::stddev();	Table	11.6

26. Reserved Words

Table::sum();	Table	11.6
Table::sum_square();	Table	11.6
Table::tabulate(double);	Table	11.2
Table::var();	Table	11.6
terminate()	Process	3.5
triangular(double mn, double mx, double mode)	Random	18.1
uniform(double mn, double mx)	Random	18.1
uniform_int(double mn, double mx)	Random	18.1

26. Reserved Words

26. Reserved Words