Contents

1 Simulation Features of OOSimL 5

2 Entities and Objects 5

3 Manipulating Active Components 6
   3.1 Defining Processes in a Simulation Model 6
   3.2 Setting Up a Simulation 8
   3.3 Start and Run a Simulation 9
   3.4 The Simulation Time 9
   3.5 Start a Process 10
   3.6 Schedule a Process 10
   3.7 Reschedule a Process 11
   3.8 Delay a Process 11
   3.9 Hold a Process 12
   3.10 Checking Idle State of a Process 12
   3.11 Get the Name of a Process 13
   3.12 Process Priorities 13
   3.13 Terminating a Process 14
   3.14 Check for Process Termination 14
   3.15 Suspend a Process 16
   3.16 Reactivate a Process 16
   3.17 Interrupting a Process 17
   3.18 Checking the Interrupt Level 17
   3.19 Get the Interrupt Level 18
   3.20 Get Remaining Time 18
   3.21 Clear Interrupt Levels 19

4 Simulation Reporting 19
   4.1 Setting The Report Files 19
   4.2 Enabling and Disabling Report Output 20
   4.3 Writing to Reports 20

5 Handling Queues 21
   5.1 Simple Queues 21
      5.1.1 Create objects of Class Queue 21
      5.1.2 Check Queue Full 21
5.1.3 Check Queue Empty ........................................ 22
5.1.4 Length of Queue ........................................ 22
5.1.5 Insert Object Into Queue ................................. 23
5.1.6 Remove Object from a Queue ............................. 23
5.1.7 Get Object from a Queue ................................. 24
5.2 Priority Queues ........................................ 24
  5.2.1 Create Objects of Class Pqueue ......................... 24
  5.2.2 Priority Queue Length .................................. 25
  5.2.3 Check Priority Queue Empty ......................... 25
  5.2.4 Queue Length With Priority ........................... 25
  5.2.5 Insert Object Into Priority Queue ................... 26
  5.2.6 Remove Object from Priority Queue ................. 26
  5.2.7 Get Object from Priority Queue ................... 27
6 Handling Resources ........................................ 27
  6.1 General Description ........................................ 27
  6.2 Standard Resources ......................................... 28
    6.2.1 Create Standard Resource Object .................. 28
    6.2.2 Available Resource Units ............................. 28
    6.2.3 Acquire Resource Units .............................. 29
    6.2.4 Release Units of a Resource ......................... 29
  6.3 Detachable Resources ..................................... 29
    6.3.1 Create Detachable Resource Objects ................ 30
    6.3.2 Available Units in a Resource Container .......... 30
    6.3.3 Take Units from a Resource Container ................ 30
    6.3.4 Give Resource Units to Container .................. 31
7 Process Interaction ......................................... 31
  7.1 Process Cooperation ....................................... 31
  7.2 Create Cooperation Objects .............................. 31
  7.3 Length of Slave Queue .................................... 32
  7.4 Length of Master Queue ................................... 32
  7.5 Wait to Cooperate ......................................... 33
  7.6 Cooperate with Slave Process ............................ 33
8 Conditional Waiting ......................................... 33
  8.1 Creating Conditional Objects ............................. 34
  8.2 Wait Until Condition ...................................... 34
  8.3 Signal Conditional Object ................................. 35
  8.4 Length of Conditional Queue ............................. 35

© 2014 J. M. Garrido
9 Random Number Generation

9.1 Randint .......................................................... 36
  9.1.1 Creating a Randint Generator .......................... 36
  9.1.2 Seed of Random Generator .............................. 37
  9.1.3 Generate Random Number ............................... 37

9.2 Random Numbers with Negative Exponential ................. 38
  9.2.1 Creating NegExp Objects ............................... 38
  9.2.2 Generate Random Values ............................... 38

9.3 Random Numbers with Normal ................................ 39
  9.3.1 Creating Normal Generator Objects ..................... 39
  9.3.2 Generate Random Values ............................... 39

9.4 Random Numbers with Uniform ................................ 40
  9.4.1 Creating Uniform Objects ............................. 40
  9.4.2 Generate Random Values ............................... 41

9.5 Random Numbers with Triangular Distribution ............... 41
  9.5.1 Creating Triangular Distribution Generator .......... 41
  9.5.2 Generate Random Values ............................... 42

9.6 Random Numbers with Poisson ................................ 42
  9.6.1 Creating Poisson Generator Objects ................... 43
  9.6.2 Generate Random Values ............................... 43

9.7 Random Numbers with Weibull ................................ 44
  9.7.1 Creating Weibull Generator Objects ................... 44
  9.7.2 Generate Random Values ............................... 44
1 Simulation Features of OOSimL

The Object-Oriented discrete-event Simulation Language, OOSimL, was developed as an educational tool and to help system modelers design and implement simulation models using the object-oriented approach to software development. OOSimL supports the process interaction approach to discrete-event simulation. This allows higher-level simulation modeling and the flexibility of including as much detail as needed.

Programs written with OOSimL can be easily integrated with Java programs. The OOSimL compiler generates Java code. The run-time support of the language is a major enhancement of the PsimJ2 simulation package, which is a collection of Java classes.

This document is Part II of the reference manual, which presents the simulation aspects of OOSimL. For the object-oriented programming aspects of OOSimL, refer to Part I of this reference manual.

The OOSimL compiler, run-time libraries, and related documentation are freely available for educational and research purposes only. The software and documentation are copyrighted (© J. Garrido) materials and are part of the OOPsim project. The most recent versions of these materials can be downloaded with examples from the OOSimL Web page; the URL for the Web pages is as follows:

\[ \text{ksuweb.kennesaw.edu/~jgarrido/psim.html} \]

2 Entities and Objects

All major components of a system model are identified as entities, which have attributes and behavior. Some of these entities will be active entities, which have a life of their own. These active entities are called processes. The other entities are passive entities, which only exhibit behavior when requested by another entity.

In OOSimL, entities are modeled and implemented as objects. An object is an instance of a class, which is considered an entity type. A process is an active object in the simulation model, and is an instance of a process class. The behavior of an object includes a set of operations (or methods) that are carried out during the lifetime of a software model in execution. An object also includes data structures that represent its attributes. In OOSimL, an active object is implemented as a Java Thread object.

During a simulation run, all the active objects of the software model interact with each other and manipulate passive objects.

© 2014 J. M. Garrido
Typically, in a software model there are several active objects of the same process definition (class). For example, a model may include several objects of class "Server," a class defined in a simulation model. Similarly, there are several passive objects such as the queues to hold customers or other objects.

The process interaction approach to discrete event simulation is based on object-oriented modeling and programming. In an object-oriented simulation and programming language such as OOSimL, an object type is defined as a class, and an object is created as an instance of that class.

3 Manipulating Active Components

The OOSimL language include facilities that implement the mechanisms necessary to develop simulation models and to carry out simulation runs using the process interaction approach.

For constructing simulation models, the language include the following facilities:

- The definition of classes for processes.
- The starting of a simulation run that will execute for a predetermined period, called the simulation period.
- Definition of various queues that are used in the simulation models.
- Definition of resource pools as passive objects.
- The generation of random numbers, each with its own probability distribution.

A simulation model must access the classes in the OOSimL library. Every user program that implements a simulation model needs to include the following use statement at the top of the source file:

```
use all psimjava
```

3.1 Defining Processes in a Simulation Model

A simulation model must define one or more classes for active objects (processes), and then create one or more instances of each of these classes. To define a process class in a simulation model implemented in OOSimL, the simulation model must define a class with an appropriate name and specify
that the class will be used for processes (active objects). All classes defined for active objects must include the clause: \textbf{as process} in the class header.

\textit{description}

\begin{verbatim}
class \langle class_name \rangle as process is 
  private
    data declarations (attributes)
    object reference declarations
    function definitions
  public
    data declarations (attributes)
    object reference declarations
    function definitions
endclass \langle class_name \rangle
\end{verbatim}

One of the classes in a simulation model is designated as the \textbf{top class} or the main class. This class must include function \textit{main}, which starts the execution of the program.

In addition to the \textit{initializer} function, another public function must be defined in every class that represents a process in the simulation model. These two functions are:

1. The \textbf{initializer} function (also known as a constructor), which is executed when an object of the class is created. The only parameter required is a character string for the name of the process. A class may include more than one initializer functions (this object oriented facility is known as \textit{overloading}).

2. The \textbf{Main\_body} function, which is present in every process class defined in the simulation model. This function defines all the activities that are to be carried out by the process, i.e., defines the actual behavior of the objects instantiated by from the process definition. The name of this function, (\textbf{Main\_body}) is the same in all processes.

The following example has a class definition in a simulation model that defines a process class, named \textit{Server}. All instances of class \textit{Server} will behave as active objects, or processes. Class \textit{Server} includes definitions of its private features (data members and methods) and its public features.
class Server as process is
// private features
....
// public features
public
function initializer parameters serv_name of type string is
begin
    call super using serv_name // parent constructor
    ....
eendfun initializer

// Main_body is
....
eendfun Main_body
// other public members
eendclass Server

One of the internal attributes in a process is its name. This and other internal features in a process class are defined in the base class Process, a class in the simulation run-time library.

To get the name of a process, use the assign name statement. The following lines of code declare a variable, machine_name, and get the name of object machine of class Server.

define machine_name of type string
....
define machine of class Server // ref to machine object
....
assign name of machine to machine_name
....

3.2 Setting Up a Simulation

The top class of every program that implements a simulation model must declare an object reference of class Simulation.

define ⟨ ref_variable ⟩ of class Simulation

To set up a simulation, the following statement is used and includes the title for the simulation project. This define statement will normally appear in function main.

simulation title ⟨ variable_string ⟩
The following lines of code declare an object reference of class \textit{Simulation}, and set up the simulation title in the simulation model.

\begin{verbatim}
... define run of class Simulation . . . simulation title "Simple Port Model"
\end{verbatim}

### 3.3 Start and Run a Simulation

The \textbf{start simulation} statement sets up the simulation with the required simulation period. This statement will normally appear in function \textit{Main\_body} in the top class.

\[
\text{start simulation with } \langle \text{time variable} \rangle
\]

The following lines of code declare the simulation period with variable \textit{simperiod}, then start the simulation to run for the simulation period.

\begin{verbatim}
define simperiod = 345.5 of type double // simulation period ...
start simulation with simperiod
\end{verbatim}

### 3.4 The Simulation Time

The simulation time is handled internally by the simulation engine of the run-time library. All variables and objects that refer to the simulation time must be defined of type \textbf{double}. The units for time measurement are not predefined, so most models implemented use the generic \textit{time units} as a time measure. All implementations of simulation models in OOSimL can access the simulation time from the \textit{simulation clock}.

The time references used by the models are based on instances of the simulation time. The \textbf{assign simulation clock} statement gets the simulation from the simulation clock and assigns it to a variable. The general form of this statement is:

\[
\text{assign simulation clock to } \langle \text{variable name} \rangle
\]

In the following example, the statements define variable \textit{tstart} and initializes it to the current simulation time; this time is accessed from the system clock.
define tstart of type double
...
assign simulation clock to tstart

3.5 Start a Process
The start statement starts execution of the main body of a process. This command must be used after creating the process.

    start object ⟨ ref_variable ⟩
    start thread ⟨ ref_variable ⟩
    start self

The following lines of code will create then start a process represented by object reference machine of class Server.

create machine of class Server using mname
start object machine // start machine object
...

3.6 Schedule a Process
The schedule statement schedules a process, after the specified time interval, or at a specified time instant. At that instant, the process will become the running process.

    schedule ⟨ ref_variable ⟩ in ⟨ time_variable ⟩
    schedule self in ⟨ time_variable ⟩
    schedule ⟨ ref_variable ⟩ at ⟨ time_variable ⟩
    schedule self at ⟨ time_variable ⟩
    schedule ⟨ ref_variable ⟩ now
    schedule self now

The following lines of code schedule the current process at the time instant timecom.

define timecom of type double // time instant to schedule process
...
schedule self at timecom // schedule at time timecom

© 2014 J. M. Garrido
3.7 Reschedule a Process

The reschedule statement reschedules a process immediately, after the specified time interval, or at a specified time instant. At that instant, the process will become the running process.

\[
\text{reschedule}\ (\text{ref}\ \text{variable})\ \text{in}\ \langle\ \text{time}\ \text{variable}\ \rangle
\]

\[
\text{reschedule}\ \text{self}\ \text{in}\ \langle\ \text{time}\ \text{variable}\ \rangle
\]

\[
\text{reschedule}\ (\text{ref}\ \text{variable})\ \text{at}\ \langle\ \text{time}\ \text{variable}\ \rangle
\]

\[
\text{reschedule}\ \text{self}\ \text{at}\ \langle\ \text{time}\ \text{variable}\ \rangle
\]

\[
\text{reschedule}\ (\text{ref}\ \text{variable})\ \text{now}
\]

\[
\text{reschedule}\ \text{self}\ \text{now}
\]

This statement reschedules a process at the time instant timecom.

\[
\text{define timecom}\ \text{of type double} \quad //\ \text{time instant to schedule process}
\]

\[
\text{reschedule}\ \text{machine}\ \text{at}\ \text{timecom} \quad //\ \text{schedule at time timecom}
\]

3.8 Delay a Process

The delay statement makes a process wait for the specified time interval. In the mean time, the process remains active waiting for the time to become the currently running process. This schedules the process at time from the current simulation time. The specified time value is the interval from the current simulation time after which the current process will become the running process. When the keyword self is used, the current process will be delayed.

\[
\text{wait}\ (\text{ref}\ \text{variable})\ \text{for}\ \langle\ \text{time}\ \text{variable}\ \rangle
\]

\[
\text{wait}\ \text{self}\ \text{for}\ \langle\ \text{time}\ \text{variable}\ \rangle
\]

The following lines of code delay the execution of the current process for a time interval, wait_p. The process is scheduled to continue execution after this period has elapsed.

\[
\text{define wait_p}\ \text{of type double} \quad //\ \text{period to wait for}
\]

\[
\text{wait}\ \text{self}\ \text{for}\ \text{wait_p} \quad //\ \text{wait for this period to elapse}
\]
3.9 Hold a Process

The **hold** statement is similar to statement **wait**. It is used when a process is to carry out an activity for a specified interval of time. The specified time value is the interval during which the activity of the process is carried out. When the keyword **self** is used, the current process will be delayed.

\[
\text{hold } \langle \text{ref\_variable} \rangle \text{ for } \langle \text{time\_variable} \rangle \\
\text{hold self for } \langle \text{time\_variable} \rangle
\]

The following lines of code hold the execution of the current process for a period \(\text{work\_p}\).

```plaintext
define work\_p of type double // period for work activity
...
hold self for work\_p
...
```

3.10 Checking Idle State of a Process

The **idle** clause is used with an **if** statement to check if the process is idle (i.e., has been suspended). This check is useful before attempting to interrupt a process.

\[
\text{if } \langle \text{ref\_variable} \rangle \text{ is [not] idle then }
...
\text{else }
\text{... }
\text{endif}
\]

The following lines of code check if process \(\text{machine}\) is idle before interrupting it with interrupt level 2.

```plaintext
define mint\_lev = 2 of type integer
...
\text{if machine is not idle then}
  \text{interrupt machine with level mint\_lev // interrupt machine}
\text{else}
  \text{...}
\text{endif}
```

© 2014 J. M. Garrido
3.11 Get the Name of a Process

The name of a process is set when the process is created. To access the name of a process, the statement \textbf{assign name} is used. The general structure of the statement follows.

\begin{verbatim}
assign name \[ of \langle ref\ variable \rangle \] to \langle var \rangle
\end{verbatim}

For example, the following lines of code gets the name of a process referenced by \textit{machine}, assigns it to variable \textit{mach\_name}, and gets the name of the current process and assigns it to variable \textit{myname}.

\begin{verbatim}
define mach\_name of type string  // name of machine
define myname of type string    // name of self
define machine of class Server  // reference to process object
...assign name of machine to mach\_name
assign name to myprio         // get name of self
\end{verbatim}

3.12 Process Priorities

The priority of a process is an integer value and represents its relative importance in relation to other processes. In OOSimL, the highest priority value is 0. To access the priority of a process, the statement \textbf{assign priority} is used. The general structure of the statement follows.

\begin{verbatim}
assign priority \[ of \langle ref\ variable \rangle \] to \langle prio\ var \rangle
\end{verbatim}

For example, the following lines of code gets the priority of process referenced by \textit{machine}, assigns it to variable \textit{mach\_prio}, and gets the priority of the current process and assigns it to variable \textit{myprio}.

\begin{verbatim}
define mach\_prio of type integer  // priority of machine
define myprio of type integer    // priority of self
define machine of class Server   // reference to object
...assign priority of machine to mach\_prio
assign priority to myprio        // get priority of self
\end{verbatim}

To set the priority of a process, the statement \textbf{fix priority} is used. The general structure of the statement follows.

\begin{verbatim}
fix priority \langle prio\ var \rangle [ to \langle ref\ variable \rangle ]
\end{verbatim}
For example, the following lines of code sets the priority 5 to the current process, then it sets priority 1 to another process referenced by `machine`.

```plaintext
fix priority 5
fix priority 1 to machine
```

The following lines of code increase the priority of a process referenced by `machine` by 1. The code defines an integer variable `m_priority` to store the priority of the process.

```plaintext
define m_priority of type integer
...
assign priority of machine to m_priority
increment m_priority // increment current priority
fix priority m_priority to machine // set new priority
```

### 3.13 Terminating a Process

The statement `terminate`, when used with a reference variable, will terminate the process referenced. When used with the keyword `self`, it terminates the current process, the one executing the statement.

```plaintext
terminate ( ref_variable )
terminate self
```

The following lines of code get the name of the object and assigns it to variable `machname` then it terminates object `machine`.

```plaintext
...
assign name of machine to machname
terminate machine
display "Process ", machname, " terminated."
...
```

### 3.14 Check for Process Termination

The `if` statement with the `terminated` clause is used to check if the referenced process has been terminated. If the process has been terminated, it cannot be scheduled or rescheduled.

```plaintext
if ( ref_variable ) is terminated
then
...
```
endif

The following lines of code check if process `machine` has been terminated, if so, it prints the name of the process (`machname`) and the message, “terminated”.
assign name of machine to machname
...
if machine is terminated // test object termination
    display machname, " terminated."
else
    ... // schedule process
endif

3.15 Suspend a Process

The **suspend** statement suspends the referenced process. The process will remain suspended until another process reactivates it.

\[
\text{suspend ( ref\_variable )} \\
\text{suspend self}
\]

The following lines of code get the name of the process referenced by `machine`, suspend the process, then display a message with the name of the referenced process.

assign name of machine to machine_name
...
suspend machine // suspend process
display machine_name, " suspended."

3.16 Reactivate a Process

The **reactivate** statement reactivates the referenced process, which had been suspended before. The process becomes activate again immediately, or in the specified time interval from the current time. The process is then scheduled to carry out normal activities.

\[
\text{reactivate ( ref\_variable ) now} \\
\text{reactivate ( ref\_variable ) in } \langle \text{time\_var} \rangle \\
\text{resume ( ref\_variable ) now} \\
\text{resume ( ref\_variable ) in } \langle \text{time\_var} \rangle
\]

The following lines of code get the name of a process referenced by `machine`, assign it to the variable `machname`, and reactivate the process immediately.

...
assign name of machine to machname
display “reactivating”, machname
reactivate machine now // reactivate process

3.17 Interrupting a Process

The **interrupt** statement sends an interrupt signal to the specified process using the indicated interrupt level. The interrupted process is rescheduled immediately so it can take appropriate action (i.e., execute its interrupt handler routine).

```
interrupt ⟨ ref_variable ⟩ with level ⟨ variable ⟩
```

In the following lines of code, the current process interrupts process referenced by *machine* with an interrupt level of 2.

```
define interr_lev = 2 of type integer
...
interrupt machine with level interr_lev
...
```

3.18 Checking the Interrupt Level

The statement **on interrupt level** is executed by an interrupted process to check if has been interrupted with a given interrupt level; if it has, then the statements in the **actions** are executed.

```
on interrupt level ⟨ int_lev_var ⟩ do
  ⟨ actions ⟩
endo
```

In the following lines of code, the interrupted process checks if it has been interrupted with a specified interrupt level. If the value of the interrupt level is true, the process executes a sequence of statements, its interrupt handling routine. If this value is not true, either the process has not been interrupted (value zero), or it has been interrupted by another process using a different interrupt level.

```
on interrupt level 5 do
  ...
endo // statements to be executed
```
3.19 Get the Interrupt Level

This assign statement gets the status of the specified interrupt level of the process. The value returned is zero if the process has not been interrupted with this level, or the interrupt level has been cleared.

If the interrupt level is not specified, this statement gets the lowest interrupt level of the calling process. This value is zero if the process has not been interrupted, or the flag has been cleared.

```plaintext
assign interrupt level (int_lev_var) [of (ref_var)] to (var_lev)
```

In the following lines of code, the current process checks if the interrupt level 2 has been set, if so, it executes a given sequence of statements.

```plaintext
set lintlev = 2
... assign interrupt level lintlev to varlev
if varlev == 1 then
    ... // execute interrupt handler
else
    ... // statements when not interrupted
endif
```

3.20 Get Remaining Time

This assign statement gets the remaining interval left after the process has been interrupted. This is useful if the interrupted process was carrying out some task for a specified period, just before it was interrupted. When the process resumes execution (gets rescheduled), it will continue executing for the remaining period.

```plaintext
assign remaining time [of (ref_var)] to (variable)
```

The following lines of code in the interrupted process store the remaining period in variable rem_period.

```plaintext
set intlev = 2
on interrupt level intlev do
    assign remaining time to rem_period
    ...
endo
```

© 2014 J. M. Garrido
3.21 Clear Interrupt Levels

This statement clears all the interrupt levels of the process (to zero). This statement is normally used in the interrupt handling routine. The clauses in brackets are optional. If the level clause is not present, all interrupt levels are cleared.

\[
\text{clear interrupt [ level \langle \text{variable} \rangle ] [ of \langle \text{ref_variable} \rangle ]}
\]

The following lines of code in the interrupted process, clear all the interrupt levels.

```oosiml
set intlev = 2
on interrupt level intlev do
    assign remaining time to rem_period
    clear interrupt // clear all interrupt levels
    ...
    // interrupt handling
enddo
```

The following lines of code in the interrupted process clear the interrupt level specified by \( \text{intlev} \).

```oosiml
set intlev = 2
on interrupt level intlev do
    assign remaining time to rem_period
    clear interrupt level intlev // clear interrupt level
    ...
    // interrupt handling
enddo
```

4 Simulation Reporting

During a simulation run, a simulation model will normally compute and display some intermediate results. After the simulation run completes, the program computes final results, displays these and terminate.

4.1 Setting The Report Files

The statement \( \text{statistics file} \) sets the text file for the statistics report of the simulation. This statement should appear immediately after setting up the simulation (using the \text{simulation title} statement, in function \text{main}).

\[
\text{statistics file ( variable_string )}
\]
The following line of code sets the statistics report to text file with name “mymodelstat.txt.

\texttt{statistics file "mymodelstat.txt"}

Statement \texttt{trace file} sets the text file for the trace report of the simulation. This statement should appear immediately after creating the object of class \textit{Simulation}.

\texttt{trace file ( variable\_string )}

The following line of code sets the trace report to text file with name “mymodeltrace.txt.

\texttt{trace file "mymodeltrace.txt"}

\section*{4.2 Enabling and Disabling Report Output}

The statement \texttt{report} sets the flag to generate the statistics and trace reports. In a similar manner, the statement \texttt{noreport} causes no reports to be generated. These statements can appear in any part of the simulation model to stop or resume the generation of reports. The general syntax for these statements follows:

\begin{verbatim}
report ...
noreport
\end{verbatim}

\section*{4.3 Writing to Reports}

Before writing to the reporting files, trace and statistics, the \texttt{access} statement is used to gain access to the reporting files in any function. The general structure of the statements is the following:

\begin{verbatim}
access trace file
access statistics file
\end{verbatim}

To write to the report files, two output statements are provided, \texttt{tracewrite} and \texttt{statwrite}. The general syntax is:

\begin{verbatim}
tracewrite ( variable\_lit\_list )
\end{verbatim}
The following lines of code, set the flag to generate the reports, access the statistical file, then write to the file the string Average value: and the value of variable vartime.

```
statwrite { variable_lit_list }

report
access statistics file
...
statwrite "Average value: ", vartime
```

5 Handling Queues

OOSimL provides two general types of queues to store references to processes in simulation models. These are simple queues and priority queues.

5.1 Simple Queues

To use simple queues in an OOSimL program, one or more objects of class Squeue must be created. These objects are simple first-in-first-out (FIFO) queues. The OOSimL statements explained in this section are used to manipulate objects of simple queues.

5.1.1 Create objects of Class Squeue

The create statement is used for creating a simple queue requires the name of the queue and the queue size (queue capacity). This last data item is optional; if not included, the assigned default size is 1,000.

For example, the following lines of code declare and create a simple queue, cust_queue with an assigned name “Customer Queue” and size of 15 customers.

```
define cust_queue of class Squeue // declare simple queue
...
// now create simple queue
create cust_queue of class Squeue using ‘‘Customer Queue’’, 15
```

5.1.2 Check Queue Full

To check for the general states of the queue, empty or full, conditions full and empty are provided. An if statement is used with these boolean values (true or false).

© 2014 J. M. Garrido
For example, the following lines of code check if queue object \texttt{cust\_queue} (defined above) is full.

\begin{verbatim}
if cust_queue is full then
  // if queue is full execute instructions here
  ...
else
  // queue is not full
  ...
endif
\end{verbatim}

5.1.3 Check Queue Empty

In a similar manner, the \texttt{if} statement with the \texttt{empty} condition checks if a queue is empty.

For example, the following lines of code check if queue object referenced by \texttt{cust\_queue} (defined above) is empty. The process suspends itself if the queue is empty, otherwise, it proceeds to dequeue an object from the queue.

\begin{verbatim}
if cust_queue is empty then
  // if queue is empty, suspend here
  suspend self
else
  // dequeue object
  ...
endif
\end{verbatim}

5.1.4 Length of Queue

The \texttt{assign length} statement gets the current number of objects in the specified queue (current queue size). The general structure of the statement follows.

\begin{verbatim}
assign length of \langle queue\_ref \rangle to \langle variable \rangle
\end{verbatim}

The following lines of code get the current length of the queue object referenced by \texttt{cust\_queue} (defined above) and assigns it to variable \texttt{qlength}.

\begin{verbatim}
define qlength of type integer
  ...
assign length of cust_queue to qlength  // get queue length
\end{verbatim}

© 2014 J. M. Garrido
5.1.5 Insert Object Into Queue

The insert statement inserts the object specified to the tail of the queue. The inserted object becomes the new object at the tail of the queue. The size of the queue is increased by one. The general structure of the statement follows.

\[
\text{insert \{ ref\_variable \} into \{ queue\_ref \}} \\
\text{insert self into \{ queue\_ref \}} \\
\text{enqueue \{ ref\_variable \} into \{ queue\_ref \}} \\
\text{enqueue self into \{ queue\_ref \}}
\]

The following lines of code insert (enqueue) the object referenced by cust\_obj into the simple queue referenced by cust\_queue.

\[
\text{define cust\_obj of class Customer} \quad //\text{ reference customer} \\
\text{// enqueue customer process} \\
\text{insert cust\_obj into cust\_queue}
\]

5.1.6 Remove Object from a Queue

The remove statement dequeues (removes) the process at the head of from the specified simple queue. After removal, the size of the queue is reduced by one, as a result of this operation. The general form of the statement follows.

\[
\text{remove \{ object\} \{ ref\_var \} of class \{ cl\_name \} } \\
\text{from \{ queue\} \{ queue\_ref \}} \\
\text{dequeue \{ object\} \{ ref\_var \} of class \{ cl\_name \} } \\
\text{from \{ queue\} \{ queue\_ref \}}
\]

The following line of code removes (dequeues) a customer process from the head of the queue referenced by cust\_queue and assigns it to reference cust\_obj.

\[\text{remove object cust\_obj of class Customer from queue cust\_queue}\]

The reference to class Customer is necessary to specify the type of object or process dequeued.
5.1.7 Get Object from a Queue

The get statement gets a copy of the process at the head of the specified simple queue, without removing it from the queue. The general form of the statement follows.

\[
\text{get} \ [ \text{object} ] \{ \text{ref}_\text{var} \} \ \text{of class} \ \{ \text{cl}_\text{name} \} \\
\text{from} \ [ \text{queue} ] \{ \text{queue}_\text{ref} \}
\]

The following line of code gets a copy of a customer process from the head of the queue referenced by cust_queue and assigns it to reference cust_obj.

get object cust_obj of class Customer from queue cust_queue

The reference to class Customer is necessary to specify the type of object or process dequeued.

5.2 Priority Queues

Priority queues allow the simulation model to retrieve or remove objects from the queue, based on the priority of the objects or processes. As such, priority queues do not behave as FIFO queues. A priority queue is an object of class Pqueue. The OOSimL statements in this section are used to define and manipulate priority queues.

5.2.1 Create Objects of Class Pqueue

The create statement is used to create a priority queue and requires the name of the queue, the optional number of different priorities, and the queue size for all priorities, which is also optional. If the second argument is not included, the number of priorities is assigned to 300 (default). If the third argument is not included, the assigned default queue size is 1,000 for every priority.

The following lines of code declare and create the priority queue with reference name cust_queue, with queue “Customer Queue”, using 8 different priorities, and size (capacity) of 25 customers for every priority.

\[
\begin{align*}
\text{define cust_queue of class Pqueue} & \quad \text{// declare priority queue} \\
\ldots & \\
\text{// create priority queue} \\
create cust_queue of class Pqueue using ‘‘Customer Queue’’, 8, 25
\end{align*}
\]
5.2.2 Priority Queue Length

This assign statement gets the current size (total number of processes) in the specified priority queue. An integer value that corresponds to the current length of the queue is assigned to the specified integer variable. The general structure of the statement follows.

\[
\text{assign length of } \langle \text{queue_ref} \rangle \text{ to } \langle \text{variable} \rangle
\]

The following lines of code get the current length of priority queue \textit{cust_queue} (defined above) and assign this value to variable \textit{mqlength}.

\begin{verbatim}
define mqlength of type integer
...
assign length of cust_queue to mqlength  // get queue length
\end{verbatim}

5.2.3 Check Priority Queue Empty

In a similar manner as for simple queues, the if statement with the is empty clause checks if the specified queue is empty.

The following lines of code check if queue object \textit{cust_queue} (defined above) is empty. The current process suspends itself if the queue is empty, otherwise, it proceeds to dequeue an object from the queue.

\begin{verbatim}
if cust_queue is empty then
  // if queue is empty, suspend here
  suspend self
  ...
else
  // dequeue object
  ...
endif
\end{verbatim}

5.2.4 Queue Length With Priority

The assign length statement gets the number of processes of a specified priority in a priority queue, and assigns it to a specified (integer) variable.

\[
\text{assign length of } \langle \text{queue_ref} \rangle \text{ with priority } \langle \text{prio_var} \rangle \text{ to } \langle \text{variable} \rangle
\]

For example, the following line of code gets the current number of processes in the priority queue \textit{cust_queue}, with priority \textit{Lprio}, and assigns this number to variable \textit{num_proc}.

© 2014 J. M. Garrido
assign length of cust_queue with priority l_prio to num_proc

5.2.5 Insert Object Into Priority Queue

The `insert` statement inserts the specified object into the priority queue. The size of the queue is increased by one. The priority of the process is automatically taken from the object to be inserted. The general form of the statement follows.

```
insert ⟨ref_variable⟩ into ⟨queue_ref⟩
insert self into ⟨queue_ref⟩
enqueue ⟨ref_variable⟩ into ⟨queue_ref⟩
enqueue self into ⟨queue_ref⟩
```

For example, the following lines of code insert (enqueue) object `cust_obj` into priority queue `pcust_queue`.

```
define cust_prio = 3 of type integer
define cust_obj of class Customer // reference customer
...
// enqueue customer process
insert cust_obj into pcust_queue
```

5.2.6 Remove Object from Priority Queue

The `remove` statement removes (dequeues) a process from the specified priority queue. This statement removes the highest priority process from the priority queue. After removal, the size of the queue is reduced by one, as a result of this operation. This statement can also remove a process with a specified priority from the priority queue. The general form of the statement is as follows:

```
remove [object] ⟨ref_var⟩ of class ⟨cl_name⟩
    from [queue] ⟨queue_ref⟩
    [with priority ⟨prio_variable⟩]
dequeue [object] ⟨ref_var⟩ of class ⟨cl_name⟩
    from [queue] ⟨queue_ref⟩
    [with priority ⟨prio_variable⟩]
```

The following lines of code remove (dequeue) a customer process from queue `cust_queue` and assigns a reference `cust_obj` to it, given priority `cust_prio`.

```
define cust_prio of type integer
```
remove object cust_obj of class Customer from queue cust_queue
    with priority cust_prio

The reference to class Customer is necessary to specify the type of object
or process) to be dequeued. If no priority is specified, the statement removes
the highest priority process from the priority queue.

5.2.7 Get Object from Priority Queue

The get statement gets a copy of the process with the highest priority from
the specified priority queue. The general form of the statement is as follows:

\[
\text{get [ object ] } \langle \text{ ref \_var } \rangle \text{ of class } \langle \text{ cl\_name } \rangle \\
\text{from [ queue ] } \langle \text{ queue\_ref } \rangle
\]

The following lines of code gets a copy of the customer process from
queue cust_queue and assigns a reference cust_obj to it.

get object cust_obj of class Customer from queue cust_queue

The reference to class Customer is necessary to specify the type of object
in the priority queue.

6 Handling Resources

6.1 General Description

The OOsimL provides facilities for manipulating resources and for the syn-
chronization of processes that compete for and use these resources. There
are two important types of resources in OOsimL:

1. Standard resources that are accessed by processes in a mutually ex-
clusive manner by processes. These resource objects represent a finite
pool of resource units and are created with the Res class.

2. Detachable resources that are produced and consumed by different pro-
cesses. These resource objects represent infinite pools of resource units
and are created with the Bin class.
6.2 Standard Resources

This section describes the most relevant statements for resource management and process synchronization of processes that access these resources. Processes normally acquire one or more resource units and must release these units eventually.

6.2.1 Create Standard Resource Object

Creating a standard resource pool involves creating an object of class Res. This object represents a finite resource pool of a specified number of resource units.

The create statement for creating a resource pool of this class requires the name of the resource pool and the pool size. The pool size defines the total number of available resource units in the resource pool.

The following lines of code declare and create a standard resource pool, chairs, with name “Customer Chairs” and a size of 30 resource units.

```oosiml
define chairs of class Res // declare resource pool 'chairs'
...
// create resource pool
create chairs of class Res using "Customer Chairs", 30
```

6.2.2 Available Resource Units

The assign available statement gets the number of available resource units available in the resource pool and assigns it to the specified variable. The general structure of this statement follows.

```oosiml
assign available units of ⟨ ref_variable ⟩ to ⟨ var_name ⟩
```

The following lines of code get the current number of available resource units in the resource pool chairs, which was defined above, and assigns this value to variable num_res.

```oosiml
define num_res of type integer // number of resource units
...
assign available units of chairs to num_res
```

6.2.3 Acquire Resource Units

The acquire statement allows the requesting process to acquire a specified number of units from the specified resource pool. If the resource pool does
not have all the requested resource units available, the requesting process is suspended.

Thus, the statement allows the process to acquire a specified number of resource units from the specified resource pool only when there are sufficient resource units available. The general form of the statement is:

\[
\text{acquire} \ (\text{int\_variable}) \ \text{units from} \ (\text{ref\_variable}) \\
\text{request} \ (\text{int\_variable}) \ \text{units from} \ (\text{ref\_variable})
\]

The following lines of code, the process attempts to acquire 10 resource units from the resource pool *chairs* (defined above).

```
acquire 10 units from chairs
// execute instructions when resources are acquired
```

### 6.2.4 Release Units of a Resource

The *release* statement releases the specified number of resource units from the process, and returns these to the resource pool. This deallocates the specified number of resource units that the process currently holds. The statement may cause reactivation of any suspended processes that are waiting for resources. One or more of the reactivated processes can then acquire the requested resources units now available, if there are sufficient resource units.

\[
\text{release} \ (\text{int\_variable}) \ \text{units of} \ (\text{ref\_variable})
\]

The following line of code, a process releases 10 resource units from the resource pool *chairs* (defined above).

```
release 10 units of chairs
```

### 6.3 Detachable Resources

An object of a detachable resource is an infinite container, which has an available number of resource units. With detachable resource objects, a process that has removed or taken a number of resource units does not need to return these units to the resource container; the process is then said to *consume* these units. Other processes may give or insert resource items to the resource container, these processes are said to be *producers*. 

© 2014 J. M. Garrido
6.3.1 Create Detachable Resource Objects

The `create` statement for detachable resources, is used to create and initialize objects of class `Bin`, with a name (or title) and the number of initial available resource units in the resource container.

The following lines of code declare and create a detachable resource container, `cust_cont`, with name “Parts” and a size of 10 resource units.

```plaintext
define cust_cont of class Bin // declare resource container
...
// create the resource container (bin pool)
create cust_cont of class Bin using "Parts", 10
```

6.3.2 Available Units in a Resource Container

The `assign available` statement gets the number of available resource units in a detachable resource object, and assigns the number to the specified integer variable. The general form of this statement follows.

```plaintext
assign available units of ⟨ ref_variable ⟩ to ⟨ var_name ⟩
```

The following lines of code gets the current number of available resource units in the resource container `cust_cont`, which was defined above, and assigns this number to the integer variable `num_units`.

```plaintext
define num_units of type integer
...
assign available units of cust_cont to num_units
```

6.3.3 Take Units from a Resource Container

The `take` statement allows a process to remove the specified number of units from the detachable resource object. If there are sufficient units in the resource container, the process takes these units and continues normally. When the operation completes, the number of resource units in the resource container is decreased by the number of resource items taken by the process.

If there are not sufficient resource units available, the process is suspended (to wait for available items). The general form of the `take` statement follows.

```plaintext
take ⟨ int_variable ⟩ units from ⟨ ref_variable ⟩
```

© 2014  J. M. Garrido
The following lines of code lets a process attempt to take 10 resource units from the \textit{cust\_cont} container (defined above).
\begin{verbatim}
  take 10 units from cust\_cont
  // execute when resource items taken
\end{verbatim}

6.3.4 Give Resource Units to Container

The \texttt{give} statement causes the current process to insert (or give) a specified number of resource units to a detachable container object. This places a specified number of resource units into the container. The number of units in the container is then updated. The general form of the statement follows.

\begin{verbatim}
  give \{ int\_variable \} units to \{ ref\_variable \}
\end{verbatim}

In the following lines of code, a process gives 8 resource units to the \textit{cust\_cont} resource container (defined above). The number of units is stored in integer variable \texttt{num\_units}.
\begin{verbatim}
  define num\_units of type integer
  ...
  set num\_units = 8
  ...
  give num\_units units to cust\_cont
\end{verbatim}

7 Process Interaction

7.1 Process Cooperation

OOSimL supports the cooperation of processes via cooperation objects. These objects implement a synchronization mechanism for the direct cooperation of processes. This mechanism allows one or more processes to dominate and be treated as \texttt{master} processes; the other processes are treated as \texttt{slave} processes.

7.2 Create Cooperation Objects

Cooperation objects are created of class \texttt{Waitq}. The \texttt{create} statement is used to create and initialize a cooperation object with a title and an optional maximum number of priorities to use. Cooperation objects have two internal priority queues, one for the \texttt{master} processes and one for the \texttt{slave} processes.
The following lines of code declare a cooperation object, \textit{coopt\_cust}, and create the object with name “Channel A”, and with default number of priorities.

\begin{verbatim}
define coopt\_cust of class Waitq ...
create coopt\_cust of class Waitq using "Channel A"
\end{verbatim}

7.3 Length of Slave Queue

The \texttt{assign length} statement assigns the length (number of processes) in the slave queue to a specified variable. This value represents the number of slave processes waiting on the cooperation (synchronization) object. The general structure of the statement follows.

\begin{verbatim}
assign length of slave queue \langle ref\_variable \rangle to \langle var\_name \rangle
\end{verbatim}

The following lines of code get the number of slave processes waiting in the \textit{coopt\_cust} object defined above and stores the value in variable \texttt{num\_slaves}.

\begin{verbatim}
define num\_slaves of type integer ...
assign length of slave queue coopt\_cust to num\_slaves
\end{verbatim}

7.4 Length of Master Queue

This \texttt{assign} statement gets the length of the master queue (i.e., the number of master processes waiting on the cooperation object), and assigns it to the specified variable. The general structure of the statement follows.

\begin{verbatim}
assign length of master queue \langle ref\_variable \rangle to \langle var\_name \rangle
\end{verbatim}

The following lines of code get the number of master processes waiting in master queue of the cooperation object \textit{coopt\_cust} defined above, and store this integer value into variable \texttt{num\_master}.

\begin{verbatim}
define num\_master of type integer ...
assign length of master queue coopt\_cust to num\_master
\end{verbatim}

© 2014 J. M. Garrido
7.5 Wait to Cooperate

The *wait for master* statement places current process in the slave queue of the cooperation object and passivates (suspends) the process. These processes will wait for a master process. This mechanism allows a slave process to cooperate with a master process, when there is a master process available. The general form of this statement follows.

```plaintext
wait for master in ⟨ ref_variable ⟩
```

The following line of code allows a slave process to wait to cooperate with a master process using the cooperation object *coopt_cust* (defined above).

```plaintext
wait for master in coopt_cust // wait for master process
... 
```

7.6 Cooperate with Slave Process

The *cooperate* statement allows a master process to attempt cooperation with slave processes. A slave process is retrieved from the slave queue of the cooperation object, if this queue is not empty. If there is no slave process available in the slave queue, the master process that executes this statement is suspended and placed in the master queue. The general form of the statement follows.

```plaintext
cooperate with slave ⟨ ref_variable ⟩ of class ⟨ cl_name ⟩ in ⟨ ref_variable ⟩
```

The following lines of code lets a master process attempt cooperation with a slave process using the *coopt_cust* synchronization object. When the cooperation becomes possible, a slave process is removed from the slave queue, the master process continues executing normally. Otherwise, the process is suspended to wait for a slave process.

```plaintext
define custobj of class Customer
...
cooperate with slave custobj of class Customer in coopt_cust
// execute when a slave is found
```

8 Conditional Waiting

The conditional wait synchronization handles the waiting of processes for a specified condition (also called conditional waiting). Conditional objects

© 2014 J. M. Garrido
of class \textit{Condq} are used as a synchronization mechanism for the processes to evaluate the specified condition and wait if the condition is not true. These conditional objects control processes to wait in a conditional queue by priority.

8.1 Creating Conditional Objects

The \textbf{create} statement creates a conditional object of class \textit{Condq}, and is used with a title and the optional number of priorities. The default number of priorities is 250.

The following lines of code declare and create a conditional object with title “Customer Cond” and with the default number of priorities.

\texttt{define cond\_cust of class Condq}

\texttt{...}

\texttt{create cond\_cust of class Condq using "Customer Cond"}

8.2 Wait Until Condition

The \textbf{waituntil} statement causes the executing process to check the specified condition and if it is false, the process is suspended. The condition specified is a boolean expression. The suspended process is placed in the hidden priority queue of the conditional object. When the process is reactivated, it should re-evaluate the condition. If the condition is true, the executing process continues normally. The \textbf{waituntil} statement should be part of the body of a loop that will repeatedly evaluate the condition. The general form of this statement is:

\texttt{waituntil \langle condition\_var \rangle in \langle ref\_variable \rangle}

In the following lines of code, a process evaluates the condition \texttt{mcond} and synchronizes with a \textit{Condq} object, \texttt{cond\_cust} (created above).

\texttt{define mcond of type boolean}

\texttt{...}

\texttt{set mcond = false}

\texttt{while mcond not equal true do}

\hspace{1em} \texttt{// evaluate boolean expression}

\hspace{2em} \texttt{set mcond = (att1 > 3) and (att2 <= att1)}

\hspace{2em} \texttt{waituntil mcond in cond\_cust}

\texttt{endwhile}

\hspace{1em} \texttt{//}

\hspace{2em} \texttt{// continue executing when the condition is true}

\copyright 2014 J. M. Garrido
8.3 Signal Conditional Object

The `signal` statement dequeues the process or processes at the head of the queue of the condition object, and reactivates them. When a process resumes execution, it will re-test the condition and may be suspended again.

```
signal to ⟨ ref_variable ⟩
```

In the following line of code, a process signals the `Condq` object, `cond_cust` (created above).

```
signal to cond_cust
```

8.4 Length of Conditional Queue

The `assign length` statement, gets the number of processes waiting in the conditional queue of the specified `Condq` object, and assigns it to the specified integer variable.

In the following lines of code, the current process gets the number of processes waiting in the conditional queue of object `cond_cust` (defined above) and assign this number to variable `cond_num`.

```
define cond_num of type integer
...
assign length of cond_cust to cond_num
```

9 Random Number Generation

OOSimL provides several types of random number generators that use different probability distributions. The following classes provide the most relevant random number generators:

- `Randint`, for the generation of uniformly distributed random numbers between 0.0 and 1.0
- `NegExp`, for the generation of random numbers from a negative exponential distribution
- `Poisson`, for the generation of random numbers from a Poisson distribution
- `Normal`, for the generation of random numbers from a normal distribution

© 2014 J. M. Garrido
• *Uniform*, for the generation of random numbers within a specified range and using a uniform probability distribution

• *Poisson*, for the generation of random numbers from a Poisson distribution

• *Gamma*, for the generation of random numbers from a Gamma distribution

• *Geometric*, for the generation of random numbers from a Geometric distribution

• *HyperGeometric*, for the generation of random numbers from a Hyper Geometric distribution

• *Binomial*, for the generation of random numbers from a Binomial distribution

• *Bernoulli*, for the generation of random numbers from a Bernoulli distribution

• *Triangular*, for the generation of random numbers from a Triangular distribution

• *Erlang*, for the generation of random numbers from an Erlang distribution

• *Weibull*, for the generation of random numbers from a Weibull distribution

9.1 Randint

The *Randint* random number generator provides the most basic type of random number generation, a uniform distribution with numbers between 0.0 and 1.0.

9.1.1 Creating a Randint Generator

The **create** statement is used to create and initialize a random number generator object of class *Randint*. For example, the following lines of code declare and create a random number generator object, referenced by *ran_gen1*, with title “rand val” and 7 as the value of the seed.
define ran_gen1 of class Randint // reference to random generator
...
create ran_gen1 of class Randint using "rand val", 7

If the seed argument is not specified in the create statement, the random generator object uses the processor timer for the seed.

9.1.2 Seed of Random Generator

The seed statement sets the seed for the random number generator. This statement is used when the random generator object has already been created and the seed set to specified value. The general structure of this statement follows.

seed to ⟨ ref_variable ⟩

The following lines of code set the seed to 9 for the generator object ran_gen1 defined above.

seed 9 to ran_gen1

9.1.3 Generate Random Number

The generate random statement generates a random number. This statement is used for all types of random generator objects. The general form of this statement follows.

generate random value ⟨ variable ⟩ from ⟨ ref_var ⟩

For objects of class Randint, the random numbers are generated between 0.0 to 1.0. For example, the following lines of code generate a random number from the generator object ran_gen1 defined previously, and stores the random number in variable ran_value.

define ran_value of type double
...
generate random value ran_value from ran_gen1

The assign random statement may also be used to generate a random number. The general form of this statement follows.

assign random value from ⟨ ref_var ⟩ to ⟨ variable ⟩
For example, the following lines of code generate a random number from the generator object `ran_gen1` defined above, and stores the random number in variable `ran_value`.

```plaintext
define ran_value of type double
...
assign random value from ran_gen1 to ran_value
```

### 9.2 Random Numbers with Negative Exponential

The `NegExp` random number generator provides random numbers using a negative exponential distribution.

#### 9.2.1 Creating NegExp Objects

The `create` statement is used to create an initialize a random number generator object. The arguments to be supplied are the name (of type `string`), the mean value of the samples (of type `double`), and the seed (of type `long`, optional argument).

For example, the following lines of code declare and create a random number generator object `nexp_gen` with name “Arrivals”, mean of 34.5 and 7 as the value of the seed.

```plaintext
define nexp_gen of class NegExp
...
create nexp_gen of class NegExp using ‘‘Arrivals’’, 34.5, 7
```

#### 9.2.2 Generate Random Values

The `generate random` statement generates a random number, using the generator object already created. The general form of this statement follows.

```plaintext
generate random value ⟨ variable ⟩ from ⟨ ref_var ⟩
```

For example, the following lines of code generate a random number from the generator object `nexp_gen` defined above, and stores the random number in variable `ran_value`.

```plaintext
define ran_value of type double
...
generate random value ran_value from nexp_gen
```
The assign statement may also be used to generate a random number using the negative exponential distribution from the random number generator created.

For example, the following lines of code generate a random number of type double using random generator nexp_gen (created above) and stores the random number in variable r_value.

```plaintext
define r_value of type double
...
assign random value from nexp_gen to r_value
```

### 9.3 Random Numbers with Normal

The Normal random number generator provides random numbers using a normal distribution.

#### 9.3.1 Creating Normal Generator Objects

The create statement is used to create an initialize a random number generator object. For Normal distributions, the arguments to be supplied are the name (of type string), the mean value of the samples (of type double), the value of the standard deviation (of type double), and the seed (of type long, optional argument).

For example, the following lines of code declare and create a random number generator object norm_gen with title “Service time”, mean of value 12.5 and standard deviation of value 2.3.

```plaintext
define norm_gen of class Normal
...
create norm_gen of class Normal using "Service time", 12.5, 2.3
```

#### 9.3.2 Generate Random Values

The generate random statement generates a random number, using the generator object already created. The general form of this statement follows.

```plaintext
 generate random value { variable } from { ref_var }
```

For example, the following lines of code generate a random number from the generator object norm_gen defined above, and stores the random number in variable ran_value.
define ran_value of type double
...
generate random value ran_value from norm_gen

The assign random statement may also be used to generate a random number using the Normal distribution from the random number generator created.

For example, the following lines of code generate a random number of type double using random generator norm_gen (created above) and store the random number in variable ran_value.

define ran_value of type double
...
assign random value from norm_gen to ran_value
...

9.4 Random Numbers with Uniform

The Uniform random number generator provides random numbers using a uniform probability distribution.

9.4.1 Creating Uniform Objects

The create statement is used to create an initialize a random number generator object. With the Uniform distribution, the arguments to be supplied are the name (of type string), the value of the lower bound of the samples (of type double), the value of the upper bound (of type double), and the seed (of type long, optional argument).

When the lower and upper bounds supplied are of type double, the Uniform object generates numbers of type double. When the lower and upper bounds are of type long, the random numbers generated are of type long.

For example, the following lines of code declare and create a random number generator object uni_gen with title “Memory request”, lower bound of value 12.5, upper bound value 55.0, and seed of value 7.

define uni_gen of class Uniform
...
create uni_gen of class Uniform using ‘Memory request’,
    12.5, 55.0, 7
9.4.2 Generate Random Values

The `generate random` statement generates a random number, using the generator object already created. The general form of this statement follows.

\[
\text{generate random value ( variable ) from ( ref_var )}
\]

For example, the following lines of code generate a random number from the generator object `uni_gen` defined above, and stores the random number in variable `ran_value`.

```
define ran_value of type double
...
generate random value ran_value from uni_gen
```

This `assign` statement may also be used to generate a random number using the Uniform distribution from the random number generator created.

For example, the following lines of code generate a random number of type `double` using random generator `uni_gen` (created above) and store the random number in variable `ran_value`.

```
define ran_value of type double
...
assign random value from uni_gen to ran_value
```

9.5 Random Numbers with Triangular Distribution

The `Triangular` random number generator provides random numbers using a triangular probability distribution.

9.5.1 Creating Triangular Distribution Generator

The `create` statement is used to create an initialize a random number generator object. With the Triangular distribution, the arguments to be supplied are the name (of type `string`), the value of the minimum or lower bound of the samples (of type `double`), the intermediate value, the value of the maximum or upper bound (of type `double`), and the seed (of type `long`, optional argument).

When the lower, intermediate, and upper bounds supplied are of type `double`, the Triangular object generates numbers of type `double`. When the lower, intermediate, and upper bounds are of type `long`, the random numbers generated are of type `long`.

© 2014 J. M. Garrido
For example, the following lines of code declare and create a random number generator object \texttt{triang\_gen} with title “Machine time”, lower bound of value 12.5, intermediate value 24.5, upper bound value 55.0, and seed of value 7.

\begin{verbatim}
define triang\_gen of class Triangular
...
create uni\_gen of class Triangular using "Machine time",
  12.5, 24.5, 55.0, 7
\end{verbatim}

### 9.5.2 Generate Random Values

The \texttt{generate random} statement generates a random number, using the generator object already created. The general form of this statement follows.

\begin{verbatim}
generate random value \langle variable \rangle from \langle ref\_var \rangle
\end{verbatim}

For example, the following lines of code generate a random number from the generator object \texttt{triang\_gen} defined above, and stores the random number in variable \texttt{ran\_value}.

\begin{verbatim}
define ran\_value of type double
...
generate random value ran\_value from triang\_gen
\end{verbatim}

This \texttt{assign} statement may also be used to generate a random number from a random number generator created.

For example, the following lines of code generate a random number of type \texttt{double} using random generator \texttt{triang\_gen} (created above) and store the random number in variable \texttt{ran\_value}.

\begin{verbatim}
define ran\_value of type double
...
assign random value from triang\_gen to ran\_value
\end{verbatim}

### 9.6 Random Numbers with Poisson

The \texttt{Poisson} random number generator provides random numbers using a Poisson probability distribution.
9.6.1 Creating Poisson Generator Objects

The `create` statement is used to create an initialize a random number generator object. With the Poisson distribution, the arguments to be supplied are the name (of type `string`), the value of the mean of the samples (of type `double`), and the seed (of type `long`, optional argument). The Poisson object generates numbers of type `long`.

For example, the following lines of code declare and create a random number generator object `poiss_gen` with title “Batch size”, mean value 24.5, and seed of value 7.

```plaintext
define poiss_gen of class Poisson ...
create poiss_gen of class Poisson using "Batch size", 24.5, 7
```

9.6.2 Generate Random Values

The `generate random` statement generates a random number, using the generator object already created. The general form of this statement follows.

```plaintext
generate random value ⟨variable⟩ from ⟨ref var⟩
```

For example, the following lines of code generate a random number from the generator object `poiss_gen` defined above, and stores the random number in variable `size_value`.

```plaintext
define size_value of type long ...
generate random value size_value from poiss_gen
```

This `assign` statement may also be used to generate a random number from a random number generator created.

For example, the following lines of code generate a random number of type `long` using random generator `poiss_gen` (created above) and store the random number in variable `size_value`.

```plaintext
define size_value of type long ...
assign random value from poiss_gen to size_value ...
```
9.7 Random Numbers with Weibull

The Weibull random number generator provides random numbers using a normal distribution.

9.7.1 Creating Weibull Generator Objects

The create statement is used to create an initialize a random number generator object. For Weibull distributions, the arguments to be supplied are the name (of type string); beta, the scale parameter; alpha, the shape parameter (both of type double); and the seed (of type long, optional argument).

For example, the following lines of code declare and create a random number generator object weib_gen with title “Failure time”, beta parameter 2.5, and alpha parameter 5.3.

```c
define weib_gen of class Weibull
... create weib_gen of class Weibull using "failure time", 2.5, 5.3
```

9.7.2 Generate Random Values

The generate random statement generates a random number, using the generator object already created. The general form of this statement follows.

```c
generate random value ⟨variable⟩ from ⟨ref var⟩
```

For example, the following lines of code generate a random number from the generator object weib_gen defined above, and stores the random number in variable fail_value.

```c
define fail_value of type double
... generate random value fail_value from weib_gen
```

The assign random statement may also be used to generate a random number using the Weibull distribution from the random number generator created.

For example, the following lines of code generate a random number of type double using random generator weib_gen (created above) and store the random number in variable fail_value.

```c
define fail_value of type double
...
assign random value from weib_gen to fail_value

...