```
process AIRPLANE
     call TOWER giving GATE yielding RUNWAY
     work TAXI.TIME (GATE, RUNWAY) minutes
     request 1 RUNWAY
     work TAKEOFF.TIME (AIRPLANE) minutes
     relinquish 1 RUNWAY
end " process AIRPLANE
```

```
process AIRPLANE
     call TOWER giving GATE yielding RUNWAY
     work TAXI.TIME (GATE, RUNWAY) minutes
     request 1 RUNWAY
     work TAKEOFF.TI
     relinquish 1
end " process AI
```

Simscript II.5

Since 1962

SIMSCRIPT II.5® Simplified

CACI

This document was produced by

Professor Tony Vignaux
Victoria University
New Zealand
Tony.Vignaux@mcs.vuw.ac.nz

Updated November 2002 by CACI

If there are questions regarding the use or availability of SIMSCRIPT II.5 product, please contact CACI at any of the following addresses:

For product Information contact:

CACI Products Company	CACI Worldwide Headquarters
1011 Camino Del Rio South, suite 230	1100 North Glebe Road
San Diego, California 92108	Arlington, Virginia 22201
Telephone: (619) 542-5224	Telephone (703) 841-7800
www.caciasl.com	www.caci.com

For technical support contact:

Manager of Technical Support
CACI Products Company
1011 Camino Del Rio South #230
San Diego, CA 92108

Telephone: (619) 542-5224

simscript@caci.com

TABLE OF CONTENTS

Abstract

SIMSCRIPT II.5 is a powerful simulation language in active use for major simulations, particularly in engineering applications. These notes illustrate just enough of the features so that simple simulations programs can be written.

1. Introduction

SIMSCRIPT II.5 is a powerful, free form, English-like, general-purpose simulation programming language. It supports the application of software engineering principles, such as structured programming and modularity, which impart orderliness and manageability to simulation models." [(Russell)(1983)] is available for a range of computers including PCs running Windows and NT and also Unix workstations.

These notes introduce only a few features of , enough for simple simulations. Many features of the language are left out. You are strongly urged to refer to the more comprehensive information found in the manuals such as [(CACI83)(1997)], and [(CACI89)(1989)], and other books such as [(CACI97)(1997)] and [(Russell)(1983)]. Nor do the notes show how to compile and run programs.

The first section is treated as a general-purpose computer language. It has all the programming structures that we are used to seeing in computer languages and it can be used for normal calculations just like Pascal, Java, or Python.

The second section describes the special additions that make into a simulation language. These include simulation time, processes, and resources.

In the third section the statistical routines used for generating random numbers and for monitoring simulations are described briefly.

2. Simscript II.5 Language Elements

SIMSCRIPT II.5 is a computer language with all the usual elements.

2.1 Program Structure

Every program must have a *main* program. In addition, it may contain a *preamble* and a number of *routines*.

main program
: Every program must contain one **main** program which is the main processing part.

preamble
: This contains all global declarations. It is not always required. If it is included it must be listed before the *main* program.

routines
: Such as Functions, Subroutines, Events, and Processes. These are not always required but will be needed in simulation programs.

2.2 The Main Program

The main program starts with the word *main*. All blocks finish with *end*

Example 2.1. From time immemorial, the first program that learners write is the one that prints out ``Hello World!". Here it is in SIMSCRIPT II.5. It consists of just one *main* block.

```
main
   print 1 line
   thus
Hello World!
end '' main
```

Comments begin with *two, single* quotes. The rest of the line is treated as a comment and ignored by the compiler.

Convention: The comment attached to the end (though it is not needed) indicates the name of the block or routine that is terminating. It used to be the convention that basic words are in lower case and identifiers are in upper case. You will find this in much of the documentation. It is not required.

Example 2.2 This is an illustration of a *main* program. It contains variable declarations, creation of resources, input, process activation, starting a simulation and calling a routine. Don't worry about the details now - they will be explained later. (This would not run by itself)

```
main
   define title as a text variable
   define no.of.tellers as an integer variable
   define lambda and mu as double variables
```

```
    create every teller(1)
    read no.of.tellers, lambda, and mu
    let u.teller(1) = no.of.tellers

    activate a customer now
    start simulation
    call report          " calling a subroutine
end "main
```

2.3 Variables

Variable names can be any combination of letters, digits, and periods that contains at least one letter or two or more nonterminal periods[1]. Variables should be explicitly declared with a *define variable-list* statement which specifies a list of variables as being of a given mode[2].

Example 2.3 Here we define one variable of integer mode, two of double mode and one of text mode.

```
define Next.Customer as an integer variable
define Weight, Length, and Height
           as double variables
define my.title  as a text variable
```

The *variable-list* can take a number of alternative forms. The variable names can be separated by any of the following: ``," or ``*and*" or ``*, and*". The choice is up to you and is intended to make the program more readable. See [(CACI83)(1997)]. Similarly, the word *an* (which could be *a*) in the first definition, above, is optional and is there for readability[3].

Convention: All variable names must be defined explicitly.

2.3.1 Modes

Variables can be of a number of *modes* (called *types* in other languages):

- integer
- real
- double (increased precision reals)
- text (for strings of text)
- pointer

integer, real, and *double* modes are intended for numerical calculations, *text* for string handling, and *pointer* for handling temporary entities and processes (to be considered in Section 3.2.1). At the start of a program all variables are initialized to zero or a blank value.

2.3.2 Local and Global Variables

A variable is only known within the routine it is defined in. It is a *local variable*. Variables declared in the *preamble* (Section 2.4) are *global* and are know and can be used throughout the program.

2.3.3 Arrays

SIMSCRIPT II.5 does have *arrays*. They are *defined* as follows:

Example 2.4 Defining 1- and 2-dimensional arrays.

```
define Vector as a 1-dimensional integer array
define Matrix as a 2-dimensional, real array
```

Before an array can be used its dimensions must be *reserved* (in the main block or a subprogram, *not in the preamble*). This allows one to have arrays where the number of elements is not known until the program is running[4].

Example 2.5 Reserving dimensions for the arrays defined in Example 2.4. The first line provides a 10-element vector of integers, referred to as *Vector(1)* to *Vector(10)*. The second a 5 by 7 matrix of real numbers whose elements are referred to as *Matrix(i,j)*, for example.

```
reserve Vector as 10
reserve Matrix as 5 by 7
```

2.4 The Preamble

The *preamble* is the section where *global* variables are defined. It must precede the *main* program and other routines and contain no executable statements.

To ensures that any variables not declared are flagged by the compiler the preamble must contain the line

normally, mode is undefined

Example 2.6 A *preamble* showing declarations of two processes, some resources and a number of global variables. The *tally* and *accumulate* statements set up statistical monitoring routines. Don't worry about the details at the moment. The idea is to see the sorts of declarations that occur in a preamble. It has *no executable statements*.

```
preamble
   normally, mode is undefined
   processes
      every customer has an Arrival.time
      define Arrival.time as a real variable
   resources include Teller

   define no.of.tellers, and no.of.customers
```

```
        as integer variables
    define Time.in.bank and Waiting.time
        as real variables

    tally Avg.Wait as the average of Waiting.time
    accumulate Avg.Line as the average of N.Q.Teller
end " preamble
```

2.5 The Assignment Statement

All statements start with a key word such as *let*, *add*, *subtract* (though the *let* can be eliminated if preferred).

Example 2.7 The first two lines show assignments with the *let* key word; The next shows an assignment without *let*. The next is an arithmetic assignment and the last two show what are effectively assignments using *add* and *subtract*.

```
    let Teller = 1
    let closing.time = closing.time/hours.v
    Number.of.Tellers = 10    " key word 'let' is omitted here
    let no.of.customers = no.of.customers + 1
    add 1 to no.of.customers
    subtract 1 from no.of.customers
```

2.6 Input and Output

In this introduction we will use only a simple free-format read statement and one simple and one formatted print statement. More complicated ones are available.

2.6.1 Free-form Input (read)

The free-form *read variable-list* statement is the simplest form of input. The *variable-list* can have variables of any mode. Data can be entered from the terminal (i.e. the system input file) or read from a file[5]. Be careful with variables of mode *text* as the value entered cannot contain blanks.

Example 2.8 This *read* statement assigns values to the 4 variables. Note that in , statements are not limited to a single line.

```
    read no.of.tellers, lambda, title
                and closing.time
```

Since the list of variables[6] can use both commas and the word *and* we could also write the above *read* statement like this:

```
    read no.of.tellers, and lambda,
            and title, and closing.time
```

2.6.2 Quick-and-dirty output (list)

This is used to help in debugging but should not be used in the final forms of programs in assignments. The *list variable-list* displays the values of the variables in the *variable-list* on the screen.

Example 2.9 This list statement prints the values of variable *x, lambda* and *no.of.tellers*:

```
   list x, lambda, and no.of.tellers
This results in the following output.
X =          2.0000000000
LAMBDA =          5.3300000000
NO.OF.TELLERS =          4.0000000000
```

2.6.3 Output with a Template (print)

The *print variable-list thus* statement, introduced here, is the main form of output we will use in the course[7]. It prints the values of every variable in the *variable-list* in a form specified by a *template*.

The *template* is a pattern showing how the values are to be presented. It can contain words other than the data being printed. Every variable in the variable-list will have a corresponding asterisk (*) pattern. Integer and text mode variables should be printed with the form *** (the number of * indicating how many digits or characters to print). Real and double mode variables will have a pattern including a decimal point somewhere (such as ***.** for a value with 3 digits or spaces before the decimal point and 2 after it).

Example 2.10 The general form is

```
print N lines with {variable-list}
thus
   {N lines of template
   with one group of *** or ***.**
   for every variable}
```

However you must be careful to count the exact number of lines used. One common error is to use too few lines and try and print from the next line which may be part of the program.

Convention: Leave an extra blank line after a print *statement*.

Example 2.11 This print statement prints 5 lines with values of the 7 variables or arithmetic expressions.

```
   print 5 lines with
      no.of.tellers, lambda, mu, and closing.time*hours.v,
      Avg.Line, Avg.no.of.Custs, and
      Avg.Wait*Minutes.per.day
      thus
   N= ** lambda= **.** mu= **.**
   closing time = **
   Avge line      = *.**
   Avge customers = *.**
   Avge wait      = ***.**

   " next program line here, thus leaving a blank
```

" after the print statement for safety.

2.7 Program Flow

SIMSCRIPT II.5 has all the usual program structures, such as *if-then-else* (or *if-else-endif*), *for-do-loop*, *until* and *while* loops, etc[8].

The *if* structure has no *then*, and the statement is terminated by *endif*[9]. The *else* part can be omitted but the *endif* must be retained.

Example 2.12 An example of the *if* structure:

```
if A > B
  let Maxvalue = A
  read C
else
  let Maxvalue = B
  read D
endif
```

while loops are similar to *until* loops. For all the iterative structures, the group of statements to be executed must be surrounded by a *do-loop* pair.

Example 2.13 Some examples of looping program structures. Here, in each case, 100 values are read sequentially into the vector *x()*

```
for i = 1 to 100
do
   read x(i)
loop

let i = 1
until i > 100
do
   read x(i)
   add 1 to i
loop

let i = 1
while i <= 100
do
   read x(i)
   add 1 to i
loop
```

2.7.1 Selector Phrases

The *for*, *while*, and *until* statements can be enriched by using one or more selector phrases to accept or reject a loop iteration. The *when* and *unless* phrases test the value of a logical expression and execute the inside of the loop if appropriate.

Example 2.14 Here baggage is only counted if it is over 1 except if it belongs to Vignaux.

```
for count = 1 to No.of.seats
  when Baggage(count) > 1
  unless Owner(count) = "Vignaux"
    do
       add Baggage(count) to Total
    loop
```

2.8 Routines

Routines in SIMSCRIPT II.5 are like subroutines or procedures in other programming languages. They are declared *after* the main block and *not* inside it. They are called from other subprograms (*not* the *preamble*). Of course, they may define local variables and use global variables.

Example 2.15 Here a routine is defined. It has a local variable, *i* and refers to *Num.Ambulances* which, we assume, is global and defined in the *preamble*.

```
routine Initialise
    define i as an integer variable

    read Num.Ambulances
    let i = 0
    until i eq Num.Ambulances
    do
       let i = i + 1
       create an Ambulance
       activate this Ambulance now
    loop
end " routine Initialise
```

Routines are executed using the *call* statement. For example:

Example 2.16 Executing a routine using a *call* statement:

```
call Initialise
```

2.8.1 Arguments

Routines can be given arguments and return results. The arguments supplying data are separated from those returning the result. This is done using the keywords *given* and *yielding*.

Arguments behave like local variables and their mode must be declared immediately after the routine heading. As they are local to the routine, they can be given any names you like.

Example 2.17 This routine calculates the value of Profit given the amount of Production.

```
routine Profit.Calc given Production yielding Profit
    define Production, Profit as double variables " arguments
    define Fixed, Contribution as double variables " local
```

```
   let Fixed = 10000.0
   let Contribution = 20.0
   let Profit = - Fixed + Contribution*Production
end " Profit.Calc
```

Example 2.18 The above routine would be called like this:

```
call Profit.Calc given 50000.0 yielding The.Profit
```

Instead of *given* you can enclose the *given* arguments in parentheses, *( ). Convention We usually declare and call routines with a list of given arguments enclosed in parentheses. You don't need the word given then.*

Example 2.19 For example, the above definition could be written:

```
routine Profit.Calc(Production) yielding Profit
   define Production, Profit as double variables
   " ..... lots of calculations ....
end " Profit.Calc
```

Example 2.20 We could call the routine like this:

```
call Profit.Calc(50000.0) yielding the.Profit
```

A routine finishes when the program ``drops off the end" of the routine. You can finish at another part by the *return* statement.

WARNING: does *not automatically convert integer* mode arguments to *real* or *double*. So don't forget to put the decimal point in real or double arguments when you call a routine!

2.8.2 Functions

Functions are routines that return a value and can be called as part of an arithmetic expression. The value is returned by using the *return with* or the *return()* statement. They are *not* executed using the *call* statement like *routines* but are used within expressions to return a value. *They must be defined as functions in the preamble*. In this definition you can set the mode and you can (and should) state the number of given arguments.

Remember that the arguments and any other local variables are not recognized outside the function. The only communication with the *main* and other routines is via the arguments in the *call* command and the return value[10].

Of course there are also many pre-defined functions, which you do not have to define yourself.

Example 2.21 We define a *Normal.fn*. In the *preamble* we define it as a *double function* and specify the number of arguments:

```
preamble
   normally, mode is undefined
   define Normal.fn as a double function
                given 2 arguments
end " preamble
```

Example 2.22 The function code would appear after the *main* block and would look something like this:

```
function Normal.fn (Mean, Std.Dev)
   define Mean and Std.Dev as double variables
   ...
   return(XXX) " some arithmetic result
end " Normal.fn
```

Example 2.23 It would then be called like this:

```
let x = Normal.fn(10.0,2.1)
      " note the decimal points for doubles !
```

Here is an example of a complete program with a *preamble* and a *function* that multiplies two variables.

Example 2.24 The *multiply* function is declared in the preamble and defined later. It is called in the main block.

```
preamble
   normally, mode is undefined
   define x,y, and z as double variables
   define multiply as a double function given 2 arguments
end "preamble

main
   read x,y
   let z = multiply(x,y)
   print 1 line with x,y,z thus
****.** times  ****.** is *******.**
end " main

function multiply(a,b)     " Definition of the function
   define a,b as double variables
   return (a*b)
end " multiply
```

2.9 Predefined Elements

There are many pre-defined functions, routines, variables and constants[11]. You do not have to define these. Pre-defined elements have special suffices, for example: *.f* for *functions*, *.r*, for *routines*, *.v* for *variables*, and *.c* for *constants*. Some examples are:

Function	abs.f(arg)	absolute value of *arg*
Routine	date.r	the current date
Variable	hours.v	hours per day
		value of □
Constant	pi.c	

3. Simulation with SIMSCRIPT II.5

We now look at some of the special facilities provides for the simulation programmer. They include *entities*, *sets*, *processes*, *resources*, and, importantly, ways of recording and elapsing simulation time.

3.1 Simulation Time

All discrete-event simulation programs have the simulation time maintained in a software clock. In this is called

time.v

time.v is a double variable. It is sed to record and print out the current simulation time particularly in controlling the simulation and in producing *traces* of its operation[12]. Do not attempt to alter *time.v* yourself.

Time is measured in *units*[13]

During the running of a simulation program, time steps forward from one *event* to the next. An event occurs whenever the state of the simulated system changes. For example, an arrival of a customer is an event. So is a departure.
Execution of this timing mechanism does not start until the following statement appears in the *main* block of the program:

start simulation

The simulation then starts, the timer routine seeking the first scheduled event[14]. It continues to run until there are no further events to execute. This is the usual method of ending a simulation.

More statements can be executed after the simulation (like the *call Report* in Example 3.1).

The program can be terminated using the stop statement.

Example 3.1 This shows only the *main* block in a simulation program. Activating the *PoissonProcess* has the effect of scheduling at least one event[15]. The *start simulation* will make the programme jump to that event When the simulation ends the *Report* routine is called.

```
main
   call Initialise
   activate a PoissonProcess now

   start simulation
   call Report
end " main
```

3.2 Entities

Before we look at *processes* we must look briefly at *entities*. An *entity* is an element of the system being simulated that can be individually identified and processed[16]. In they can be either *permanent* or *temporary*. They are defined in the *preamble* using either an *include* or an *every* declaration statement (*every* statements will be dealt with in Section 3.2.2.)

Example 3.2 Defining some entities.

```
preamble
   normally, mode is undefined
   permanent entities
      include Teller, Computer
   temporary entities
      include customer, Task
   ....
end " preamble
```

We will not say much about permanent entities in this simplified version of the language though they will be used, in the form of *resources* (see Section 3.5).

3.2.1 Creating and using Temporary Entities

Temporary entities must not only be *defined* in the *preamble*, they must be *created* before they can be used.

Example 3.3 Assuming that the *customer* has been defined as a temporary entity we can *create* it as follows[17]:

```
create a customer
```

When a temporary entity, say *customer*, is created:

- a section of memory is allocated to hold its attribute values (see section 3.2.2 to find out what attributes are) and
- a single global variable *customer* points to this memory space.

When we create another *customer*, the same global integer variable *customer* then points to the location of the new entity. (c.f. the *new* statement in many modern languages). We would not use that global value but define and use a *pointer* variable to refer to the new entity. So it is better (and our convention) to *create* temporary entities in the following form (where *PV* is previously defined as a *pointer* variable). *PV* can then be used to refer to the new entity.

Example 3.4 Creating a customer with a pointer that can be used to refer to it.

```
create a customer called PV
```

An entity can later be *destroyed*. It is best to do this by way of the pointer variable. Just as the *create* statement needed the entity as well as the pointer, so does the *destroy* statement.

There is an important difference between the *create* and *destroy* statements. We *create* ***a*** *customer called PV* but *destroy* ***the*** *customer called PV*.

Example 3.5 Here we define pointer variables *Fred* and *Valerie* which are used to access the two different temporary *customer* entities. We then get rid of *Fred* using the *destroy the* command. Note the different usage of *a* and *the* in the two statements.

```
define Fred, Valerie as pointer variables
  ...
create a customer called Fred
create a customer called Valerie
  ...
destroy the customer called Fred
```

3.2.2 Attributes of Entities

An *attribute* is a property of an entity that conveys extra information about it. Attributes can be used to identify a sub-class of entities (e.g. red cars) or values for an individual entity (a customer number) or to control its behavior. The programmer can define attributes for the entities in the program.

We define an entity with attributes by the *every* statement in the *temporary entities* section of the *preamble*, instead of the *include* statement. You state what attributes the entity has and define their modes.

Convention: The modes of attributes of entities are defined immediately they are specified and are given names that start with a prefix indicating the entity that they belong to.

Example 3.6 Here we define temporary entities without (*Task*) and with (*customer*) attributes:

```
preamble
   normally, mode is undefined
   temporary entities
     include Task                 "(no attributes)
     every customer has a cu.Arrival.time
       define cu.Arrival.time as a double variable
   ...
end " preamble
```

Later in the program you can use the attribute, indexed by the global entity name or a pointer variable

Example 3.7 Using the *Fred* pointer variable to set the *Arrival.time* for that particular *customer*.

```
main
   define Fred as a pointer variable
   ...
   create a customer called Fred
   let cu.Arrival.time(Fred) = 11.4
   ...
end " main
```

3.3 Sets

A *set* is an ordered list of *entities*, like a queue. Sets must be defined in the *preamble* and must have an *owner*. The owner is often another entity. In some cases this is not appropriate so we can specify that a general-purpose owner, the *system*, can own a set. When you define an *entity* in the preamble, you must specify what sets it can belong to and what sets it owns.

Example 3.8 Here, each of the *Job entities* has its own list of *Task*s to be done, the *Task.list* held as a *set*. The *Job* can be put on (it *belongs to*)[18] the single *Job.queue* which is owned by the *system*.

```
preamble
   temporary entities
     every Task
        has a Tk.duration,
        and belongs to a Task.list
        define Tk.duration as a double variable
     every Job
        has an Jb.arrival.time,
        owns a Task.list, and
           belongs to a Job.queue

     the system owns the Job.queue
  ...
end " preamble
```

Sets (queues), if not further defined, are assumed to have a normal FIFO (*First-in, first-out* or *First-come, first served*). Otherwise they can be defined as being a *LIFO* set. Alternatively a set can be ranked by combinations of attributes of the entities that can belong to it. This is specified when the *belongs* phrase is used (See the *define .. as a set ...* in Example 3.9).

Example 3.9 Here the *Task.list* contains *Tasks* filed in order of low values (ascending order) of *Tk.duration*.

```
   temporary entities
     every Task
        has a Tk.duration,
        and belongs to a Task.list
        define Tk.duration as a double variable
        define Task.list as a set
            ranked by low Tk.duration
```

Entities are put into and removed from sets by statements in the routines of the program. They are added to a set using the *file* statement (See line 5 in Example 3.10), and taken out using the *remove* statement (Line 7 in Example 3.10). The entities are filed in the appropriate order automatically.

Sets have a number of standard attributes, automatically defined and updated for you (see Section 3.6). A most useful one is the number of *entities* in the set. For a set called *Q*, the number of entities it holds is *N.Q*. This is *automatically* updated whenever entities are added to or removed from the set.

You can test if a set is empty or not using the forms *Q is empty* or *Q is not empty*. You can also ask if a particular entity, say *E*, is in the set using: *E is in Q* or *E is not in Q*.

Example 3.10 (following on from Problem 3.8) Here we create a *Job100* and a *Task* which is put onto *Job100*s *Task.list*. The print statement (6) outputs the number in the *Task.list*. We can *remove* the task from the set (7). The particular statement used does not destroy the task and it can be accessed using *tt.*

```
1) define Job100, and tt as pointer variables
  ...
2) create a Job called  Job100
3) create a Task called tt
4) let Tk.durationn(tt) = 100.0
5) file tt in the Task.list(Job100)
  ...
6) print 1 line with N.Task.list(Job100) thus
   Number in task list is ****
  ....
7) remove the first tt from  the Task.list(Job100)
 ...
8) if the Task.list(Job100) is empty
9)     print 1 line thus
10) Set is empty
11)
12)endif
```

3.4 Processes

We now come to the main tool for discrete-event simulation. A *process* is a sequence of events that describes the experience of an entity as it lives its lifetime. For example, a message turns up in a computing network; it makes transitions between nodes, waits for service at each one, and eventually leaves the system. All these are events.

In a process is represented as an *entity* associated with a *process routine* that describes the life-cycle. This entity is a *temporary entity* (but is defined in a separate section of the *preamble* as a *process*). A *process* can have attributes. Different individual processes (e.g. individual customers) are created as the program runs. The *routine*, which describes the sequence of events, is called the *process routine* and is written in the program as a routine with the same name as the entity.

3.4.1 Defining a Process

A process entity is defined with or without attributes in a *processes* section of the *preamble*. *Processes* can own and belong to *set*s.

Example 3.11 Here we define a *message* process (with no attributes), a *clock*, and a *customer* process (with attributes).

```
preamble
  normally, mode is undefined
  processes
    include message  " a process with no attributes

    every clock has a cl.int
      define cl.int as a real variable

    every customer has an cu.Arrival.time
      define cu.Arrival.time  as a real variable
end " preamble
```

3.4.2 Process Routine

The process routine is, like other routines, declared after the *main* block but uses the keyword *process*. Its name must be that of the process defined in the *preamble*. The process starts executing the routine when it is activated.

Example 3.12 A process routine for a ticking clock. *int* is the time interval between ticks. It would be defined as an attribute of the process entity and an argument of the process routine.

```
process clock(int)
   define int as a double variable
   define i as an integer variable
   for i = 1 to 10
   do
      print 1 line with time.v
      thus
***.*** tick
      wait int units
   loop
end " clock
```

3.4.3 Creating a Process Entity

A new process entity appears when the process is created or activated. To create and start a new process in one step, use the *activate* statement. Pointer variables can be used to reference such new processes.

Example 3.13 We activate two *message*s immediately. the word *a* indicates that we are both creating and activating new messages (they have no attributes). The word *now* implies no simulation delay before the process starts.

```
   activate a message now
   activate a message called mmm now
```

It is also possible to give values to the attributes of the process *before* it is activated. This is done by first creating it, setting the attribute values and then activating it.

> *Example 3.14* Here the *customer* is created first, given an attribute value, then activated. The word *the* is used to indicate we are not creating a new *customer* when we *activate*. The attribute *cu.Arrival.time* here records the time the customer is created.

```
create a customer called Fred
let cu.Arrival.time(Fred) = time.v
activate the customer called Fred now
```

Processes do not have to be activated immediately.

> *Example 3.15* Here we activate a *Job* to start at some (random) time in the future and activate a *customer* (to be called *Fred*) immediately, and a bus when *time.v* becomes 120.0 time units. The word *a* indicates that these processes are being created.

```
   activate a Job in exponential.f(13.33, 2) hours
   activate a customer called Fred  now
   activate a Bus called b at 120.0 units
```

3.4.4 Activating a Process with Attributes and Arguments

A process with *attributes* may have a corresponding routine with *arguments* that correspond in mode *but not name* to the attributes of the process entity.

> *Example 3.16* For the customer process definition shown in Example 3.11, a corresponding process routine fragment might be:

```
process customer(ArrTime)
   define ArrTime as a real variable
   ...
end " customer
```

We can then activate the process and at the same time assign a value of 13.3 to the attribute cu.Arrival.time by either of the following methods:

1. Assign an cu.Arrival.time with the value of the process argument, *ArrTime*, when the routine starts.

   Example 3.17 Process argument used[19]:
   ```
   activate a customer called Fred given 13.3 now
   ```

2. Set the value of the attribute and then activate the process. This ignores the process argument:

   Example 3.18 3-step process:
   ```
      create a customer called Fred
      let cu.Arrival.time(Fred) = 13.3
      activate the customer called Fred now
   ```

However, a *process attribute*, such as *cu.Arrival.time*, and a *process routine argument*, such as *ArrTime*, cannot have the same name[20].

3.4.5 Elapsing Time in a Process

Within a *process routine* the process can *wait* or *work* for some time interval measured in *units*. (It can also *request* and *relinquish* resources as described later in Section 3.5). In this way it can simulate the elapsing of time.

Example 3.19 The process will hold for a random time with mean 10.0 units.

```
  work exponential.f(10.0, 3) units
```

wait and *work* are effectively synonymous but work is more appropriate if a resource is being used.

A process disappears when it is destroyed or when it runs off the end of its routine. *Convention: we do not usually destroy processes. instead, allow them to run off the end of the routine.*

3.4.6 A Complete Program Using *wait* Commands

Example 3.20 This program simulates a firework with a time fuse. It contains a preamble to define the firework process. I have put in a few extra *wait* commands

```
preamble
  normally mode is undefined
  processes
    include firework
end " preamble

main
  activate a firework in 20.0 units
  start simulation
end "main

process firework
  define i as an integer variable
  print 1 line with time.v thus
*****.** firework activated

  wait 10.0 units
  for i = 1 to 10
  do
    wait 1.0 unit
    print 1 line with time.v, and i thus
*****.** tick **

  loop
  wait 10.0 units
  print 1 line with time.v thus
*****.** Boom!!
end " process firework
```

The output from the program in Example 3.20 is :

```
  20.00 firework activated
  31.00 tick  1
```

```
32.00 tick  2
33.00 tick  3
34.00 tick  4
35.00 tick  5
36.00 tick  6
37.00 tick  7
38.00 tick  8
39.00 tick  9
40.00 tick 10
50.00 Boom!!
```

One useful program pattern is the *generator* (See Example 3.21). This is a process that generated events or activates a number of other processes as a sequence - it is a source or generator of other processes. Random arrivals are generated using such a process.

Example 3.21 The generator pattern. A *process* to generate a series of *customer*s to arrive at intervals of 10.0 units of time. To achieve ``random'' arrivals of *customer*s the *wait* statement should use an *exponential* random variate instead of, as here, a constant 10.0 value.

```
process Generator
   while time.v < Finish.time
   do
      activate a customer now
      wait 10.0 units
   loop
end " Generator
```

3.4.7 Interactions of Processes

The various states that a process can exist in are shown in this diagram:

An executing *process* can *suspend* itself and can be sent a *reactivate* command by another process.

Example 3.22 The process itself would say

```
suspend
```

and (some other process) would *reactivate* using the form:

Example 3.23 reactivation by a second process.

```
reactivate the customer called Fred now
```

3.4.8 Interruptions

A *pending* process (one that is waiting for an event to occur) can be *interrupted* and later *resume*d by another routine.

Example 3.24 Here we are in some routine (not the *customer* routine. The *customer* called *Fred* is interrupted, filed in a *set*. It is held there for 20.0 units and then resumed.

```
   interrupt the customer called Fred now
   file Fred in Interrupted.Set
   wait 20.0 units
   remove the  first customer called Fred
        from the Interrupted.Set
   resume the customer called Fred
```

Example 3.25 In this program example the *Arrival.Generator* is interrupted by the terminating process *Close.Doors*. This routine prints out a report and then stops the program, even though some processes have not yet finished. First we look at the *preamble* and *main* blocks.

```
preamble
   normally mode is undefined
   processes
     include Arrival.Generator,and  Close.Doors
     every customer has a cu.number
       define cu.number as an integer variable

   define Day.Length, Mean, sojourn
                  as real variables
   define count.departs, and count.arrivals
      as integer variables
end " preamble

main
   call Read.Data  "reads 3 global variables
   count.departs = 0
   count.arrivals = 0
   activate an Arrival.Generator now
   activate a Close.Doors in Day.Length units
   start simulation  " it starts operating here
   call Report
end " main
```

Example 3.26 Following on from Example (3.25) we list the processes and routines.

```
process Arrival.Generator
   define i as an integer variable
   for i = 1 to 1000000
   do
     wait exponential.f(Mean,1) units
     activate a customer(i) now
     add 1 to count.arrivals
   loop
end " Arrival.Generator

process customer(id)
   define id as an integer variable
   print 1 line with time.v, and id thus
***.*** customer *** arrives

   wait exponential.f(sojourn,2) units
   print 1 line with time.v, and id thus
***.*** customer *** departs
   add 1 to count.departs
end "customer

process Close.Doors
   interrupt Arrival.Generator
   call Report
   stop
end "Close.Doors

routine Read.Data
   read Day.Length, Mean, and sojourn
end " Read.Data

routine Report
   print 2 line with count.arrivals
      and count.departs thus
***** customers arrived
***** customers departed
end " Report
```

Example 3.27 An example of the trace output from a run of this program is as follows:

```
   .078 customer   1 arrives
  1.803 customer   2 arrives
  3.219 customer   3 arrives
  3.242 customer   4 arrives
  3.619 customer   3 departs
  4.643 customer   5 arrives
  4.803 customer   6 arrives
  5.146 customer   1 departs
  5.222 customer   7 arrives
  5.404 customer   8 arrives
  5.654 customer   8 departs
  5.847 customer   7 departs
  6.315 customer   9 arrives
  6.860 customer   2 departs
  7.285 customer   6 departs
  7.765 customer  10 arrives
  7.882 customer   9 departs
   10 customers arrived
    7 customers departed
```

3.5 Resources

A *resource* models a congestion point where there may be queuing. For example in a manufacturing plant, a *Task* (modeled as a *process*) needs work done at a particular sort of *Machine* (modeled as a *resource*). If not enough *Machines* are available; the *Task* will have to wait until one becomes free. The *Task* will then have the use of a *Machine* for however long it needs. It is not available for other *Tasks* until it is released. These actions are all automatically taken care of by the resource.

A *resource* can have a number of different *types*. There may be three types of *Machine resource*, perhaps, lathe, cutter, and polisher. A *Ship* may be a tanker, a general cargo ship, or a ferry. Each different type of a resource has a number of *units*. This records how many of that type there are initially.

In a *resource* is modeled as a special kind of *permanent entity*. A process gets service by *request*ing and, when it is finished, by *relinquish*ing the resource. A *resource* maintains a *set* (a queue or list) of processes using units of each type the resource and another *set* of processes waiting for units of it. These *sets* are defined and updated automatically. Resources are defined in the *preamble* in a special section labelled *resources*. A resource can have user-defined attributes.

> *Example 3.28* Here *Teller* is defined as a resource without user-defined attributes. *Ma.name* is a user-defined attribute for the resource *Machine*.

```
resources
   include Teller
   every Machine has a Ma.name
       define Ma.name as a text variable
```

A *resource* also has standard attributes defined automatically (See Section 3.6). These record how many units of each type of the resource are free, how many processes are waiting for it, etc. For example, the number of types of a *resource R* is held in the variable *N.R*. Type *i* is referred to as *R(i)* where *i = 1 to N.R*. Each type has a number of *units*. The number of *units* of type *i* is referred to as *U.R(i)*.

Once a *resource* has been defined in the *preamble* it must have storage allocated for its attributes in the *main* block[21]. First you must specify the number of types (*N.R*); then you *create* all the types (*create every R*), then you specify how many units of each type (*U.R(i)*).

> *Example 3.29* A group of *Machines* (a *resource*) might have 2 different types (*N.Machines = 2*).There can be different numbers of units of each type:

```
let N.Machines = 2   " the number of TYPES

create every Machine  "creates storage

let U.Machine(1) = 4 " the number of UNITS of type 1
let U.Machine(2) = 1 " the number of UNITS of type 2
```

You can get information about the set of *processes* that are using units of the *resource* and those of *processes* that are waiting. For *resource R* the *set* using units of type *i* is known as *X.R(i)*; the *set* waiting for it as *Q.R(i)*. Then, of course, since *Q.R(i)* is a *set*, we can find out how many *processes* are waiting from the set attribute *N.Q.R(i)*.

> *Example 3.30* for the *Machines*, in Example3.29, look at type 1 Machines only (there are 4 of them)

```
Q.Machine(1)   " the  set of processes waiting for a
             " type 1 Machine
N.Q.Machine(1) " the number of processes waiting
X.Machine(1)   " the set containing resources using
             " units of a type 1 Machine
N.X.Machine(1) " the number of processes using units
```

3.5.1 Using Resources

The process routines will contain statements to interact with the resource. These are the *request* and the *relinquish* statements. To use a resource for a time a process will *request* a number of units of the particular type. If they are available they are allocated to the process, which then holds them until releasing them. If they are not available, the processes will he held on the waiting queue until some come free. The process will suspend its operation until it receives its request. See Example (3.31).

On finishing with the resource the process must release it using the *relinquish* statement, saying how many units of what type are to be released[22].

Example 3.31 Here the *Job requests*, and if necessary waits for, one unit of a *Machine* of type 2. On acquisition it then holds it while it *work*s for a random time (exponentially distributed, mean 20.0) units and then *relinquish*es it again.

```
process Job
   request 1 Machine(2)
     work exponential.f(20.0,3) units
   relinquish 1 Machine(2)
end " Job
```

3.6 Standard Attributes

Entities, sets, processes and resources have predefined standard attributes in addition to those defined by the user. These are listed in the manuals. Only a few are tabulated here:

Some automatically generated attributes			
routines and variables			
Entities	variable	entity	global variable
	variable	N.entity	no of entities in class (only permanent entities)
Processes	attribute	time.a	next scheduled entry time for the process
Resources	set	Q.resource	set of processes waiting for this resource
	set	X.resource	set of processes using this resource
	attribute	U.resource	number of idle units
Sets	attributes	F.set	first entity in set
	of owner	L.set	last entity in set
	entities	N.set	number of entities in set
Sets	attributes	P.set	pointer to predecessor in set

	of member	S.set	pointer to successor in set
	entities	M.set	equals 1 if entity is in the set, 0 otherwise

Thus the processes waiting for a resource *R* are listed in the set *Q.R* and the variable *N.Q.R* tells you how many there are.

To list them, you would start at *F.Q.R* which is the first one waiting. If *First.one* = *F.Q.R*, the next one waiting is *S.First.one* (the successor to the first one).

4. Statistical Aspects of Simulation

SIMSCRIPT II.5 provides a number of statistical facilities. These include random variate generation with different seed streams and statistical monitoring methods. See [(CACI)(1997)][Section 5.3].

4.1 Random Number Generation

There are 10 random number *stream*s, numbered S = 1 ... 10. At any instant each stream has an integer seed value, which is updated every time the stream is called on for a random number. All the seed values are different. A stream's seed value can be found from the variable *seed.v(S)*.

The stream is updated using one of the random variable functions. All these have the stream as a parameter and update that stream one or more times whenever they are called.

Example 4.1 Generating a random variable from stream 3.
```
let S = 3
let X = random.f(S)
```

In this example, the seed value for stream 3 is updated to the next ``random" integer and the real value X = *seed.v(3)/(*2^{31} *1)* is returned as a real random variable[23]. Thus 0 < X < 1.

Thus *random.f(S)* generates apparently uniform random deviates between 0 and 1. It uses one of the 10 random number seed variables. Each seed sequence of random integers is supposedly independent of of the others.

The initial values of the seed integers (the starting seeds) from each of the *STREAMs* are initialized by at the start of a run. They are listed at the start of Appendix C of [(Russell)(1983)]. For example, *seed.v(1)=2116429302*. You can supply your own initial integer values if you want:

Example 4.2 Set the initial seed value for stream 3 to 2345.
```
 let seed.v(3) = 2345
```

Pseudo-random variables can be generated from a number of distributions[24]:

- *binomial.f(N,P,STREAM)*
- *erlang.f(Mean,K,STREAM)*
- *exponential.f(Mean,STREAM)*
- *gamma.f(Mean,K,STREAM)*
- *log.normal.f(Mean,StdDev,STREAM)*

- *normal.f(Mean,StdDev,STREAM)*
- *randi.f(Start,Finish,STREAM)*
- *uniform.f(Start,Finish,STREAM)*

These routines contain calls to *random.f(STREAM)* and use the corresponding seed stream.

> *Example 4.3* To generate a sample, *x*, from an exponential distribution with mean 20.0, using STREAM 1, one would use the following call. Notice that I have been careful to put in a decimal point into the first argument which must be real.
>
> ```
>  x  = exponential.f(20.0,1)
> ```

NOTE: You must use *real* values if a routine requires a *real* argument. For example *exponential.f(3, 1)* will not give you a sample from an exponential with mean 3. Instead, you must use the call *exponential.f(3.0, 1)*. **does NOT automatically convert arguments from *integer* to *real*.**

4.2 Statistical Monitoring

Any *global* variable can be automatically monitored by the system to calculate statistics. Little change has to be done to the program except by the addition of a special statement in the *preamble*. The two types of monitoring are *tally* and *accumulate*.

4.2.1 Tally

tally takes a sample every time the variable is changed. It is intended to monitor a number of individual observations, such as waiting times.

> *Example 4.4* here in the *preamble* we define two global variables[25] and then indicate that they are to be monitored using the *tally* statement. The *tally* statement specifies that we want to find both the *mean* and *standard deviation* of the values of the *waiting.time*.
>
> ```
> preamble
>    ...
>    define Waiting.time and Time.in.bank as real variables
>    ...
>    tally Avg.Wait as the mean ,
>       and Sd.Wait as the std.dev of Waiting.time
>    ...
> end "preamble
> ```
>
> Later in the main part of the program the mean and standard deviation of the *Waiting.time* can be used:
>
> ```
>    print 1 line with Avg.Wait and Sd.Wait thus
> Waiting time: mean = ****.***, sd = ****.***
> ```

4.2.2 Accumulate

accumulate is intended to monitor variables that are continuous in time such as the length of a queue. Although it changes discretely, there is always a queue length existing[26].

accumulate calculates the time-integrals of the value such as the average queue length.

Example 4.5 Here the queue at the *Teller* is to be monitored to determine the *average* of the number waiting for the *Teller* (held in *N.Q.Teller*).

```
preamble
  ...
  resources include Teller
  ..
  accumulate Avg.Line
    as the mean of N.Q.Teller
  ...
end " preamble
```

Then in the main part of the program we could find out the average length of the line so far:

```
  print 1 line with Avg.Line thus
The average number waiting so far is ****.***
```

tally and *accumulate* can calculate a number of different statistics: *number*, *sum*, *mean*, *sum.of.squares*, *mean.square*, *variance*, *std.dev*, *maximum*, *minimum*.

5. Acknowledgments

Students in OPRE352 and COMP349 classes have, year by year, improved these notes by their comments and questions. I am also particularly grateful to Jay Braun, the author of the *Reference Handbook*, [(CACI83)(1997)], and now with JPL, who kindly corrected a number of mistakes and suggested changes.

I will be grateful for any corrections or suggestions for improvements to the document from students, experts, or anyone accessing it over the Web. My email address is Tony.Vignaux@vuw.ac.nz.

5.1 References

[(CACI83)(1997)]
CACI83 (1997). SIMSCRIPT *II.5 Reference Handbook*. C.A.C.I, 2nd edition.
[(CACI89)(1989)]
CACI89 (1989). UNIX *SIMSCRIPT II.5 User's Manual*. C.A.C.I.
[(Law and Larmey)(1984)]
Law, A. M. and Larmey, C. S. (1984). *An Introduction to Simulation using SIMSCRIPT II.5*. C.A.C.I.
[(CACI97)(1997)]
CACI97 (1997). SIMSCRIPT *II.5 Programming Language*. C.A.C.I.
[(Pidd)(1992)]
Pidd, M., editor (1992). *Computer Simulation in Management Science*. Wiley, 3rd edition.
[(Russell)(1983)]
Russell, E. C. (1983). *Building Models with SIMSCRIPT II.5*. C.A.C.I.

5.1.1 Footnotes

[1] II.5 disregards all periods written at the end of names and numbers. Thus the names *flight...*and *flight..* become flight when the program is compiled. (SIMSCRIPT II.5 Reference Manual) You can have periods within an identifier.

[2]In the word ``mode'' is used instead of ``type''

[3]*Don't read this!* If a name is *not* explicitly declared, will declare it *implicitly* with *double* mode. Implicit declaration is strongly discouraged because it can lead to errors that are hard detect. We forbid this using a command in the preamble (see the beginning of Section 2.4).

[4]Arrays can be multiply-dimensioned and can even be ``ragged'' where the rows are not all of the same length

[5]In Unix systems, such a file can be read by redirection. For example, when running program *harbour* that is written to read from the terminal and write to the terminal, without changing the program we can make it read from *datafile* and write the results to file *results*. using: *harbour < datafile > results*

[6]See the definition of *variable list* in the Reference handbook

[7]See the documentation for others

[8]it also has a *case* statement

[9]Or *always*, *otherwise*

[10]Or in Global variables

[11]See the Reference Handbook

[12]A *trace* is an output record that lists every event and the simulation time it occurred. It is essential for debugging simulation code and can also be analyzed after a run to produce statistical results. We will be using *traces* frequently.

[13]Other time units are *days*. *hours* and *minutes*. *Convention: unless the simulation is actually operating in days, hours and minutes, the term units will be used to measure time.*

[14]At least one *process* must be activated before the simulation is started otherwise no simulation will occur

[15]See section 3.4 to find out more about processes

[16][(Pidd)(1992)] treats entities more fully.

[17]The *a* is important

[18]You can use *may belong to* or *can belong to* instead of *belongs to*

[19]I get an error if the bracketed method is used though it is documented in the Reference handbook

[20]If this sounds a bit complicated, that is because it is. For comments on this, see [(Law and Larmey)(1984)].

[21]or in another routine

[22]If you do not balance the *request* and *relinquish* pair your program may grind to a halt

[23]Because the largest integer that the seeds can take is (2^{31} 1)

[24]Yes, Valerie, there *is* a *poisson* distribution [*poisson.f(Mean,STREAM)*] but it is *never* used in simulations. Understand? Never!

[25]Remember you can only tally global variables

[26]though it may sometimes be zero