A Study of the Feasibility of Ada as a Simulation Language

by

DAVID L. KWEDER

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in the

Mathematics Program

John J. Benedetto — 3/18/92

Advisor — Date

Sally M. Hotchkiss — March 18, 1992

Dean of Graduate School — Date

YOUNGSTOWN STATE UNIVERSITY

March, 1992

Abstract

A Study of the Feasibility of Ada as a Simulation Language

David L. Kweder

Master of Science in Mathematics

Youngstown State University, 1992

The purpose of this thesis is to develop an Ada tool where capabilities are similar to those of the GPSS simulation language. The question of whether Ada is a feasible language for implementing discrete event simulations is then explored.

Acknowledgement

I would like to thank the following people for their help: Mike Newland, for his assistance in helping me format this text and in the proof reading; Karen DeMatteo, for assisting me by providing information needed to help write this thesis; Tom Wheeler, for his help and advice in writing the Ada simulation tool and finally, Professor John J. Buoni for reviewing this thesis and advising me during its writing.

Table of Contents

List of Examples

Appendix

I. The GPSS and Ada programs as well as their output for each example.

II. Listing of Ada packages used to build Sim_Ada Tool.

III. List of Ada packages used in the simulation tool.

Dedication

to Vail

Introduction

Is Ada a feasible language for implementing discrete event simulations and how does it compare to a standard simulation language, for example GPSS? In [2], Payne develops a simulation programming system for discrete event models in PL/1. Shore in [3] seems to be the first to discuss the concepts necessary to allow a programmer to construct a discrete-event simulation in Ada. The purpose of this thesis is to develop a tool written in Ada capable of performing discrete event simulations.

Chapter one is divided into two sections. The first section is a brief review of GPSS including a history of the language, features of the language, and some syntax of the language. The second half of chapter one is a brief overview of Ada, including the history of Ada, the package feature of Ada used in developing the Simulation System in Ada, the Sim_Ada tool.

Chapter two is a discussion of the Sim_Ada tool. This chapter is divided into two sections. The first section is a discussion of how the Sim_Ada tool was developed and the special features of Ada that were used in creating the tool. The second section is a discussion of how the tool is used in creating models and simulations. Appendix II contains all of the packages and Ada code used to create the Sim_Ada tool.

Chapter three is a discussion of the three examples, a grocery check-out line, an inventory control model, and a multiuser computer system. Each one of these three examples includes a discussion of the problem, results of the outcome in both GPSS and Ada, and concluding remarks. Appendix I contains the GPSS and Ada programs as well as the program output for each example.

Chapter four, the final chapter, contains a comparison and contrast of the GPSS language with the Sim_Ada tool, and some final conclusions.

Chapter 1

Discussion of Discrete Event Simulations and the Ada Language

A model is an attempt to create or recreate a real life situation or system inside a computer. We would like to make use of these models in order to make decisions about our real life system, such as improving our system, troubleshooting our system before a problem occurs, and testing different ways of implementing our system. It would seem reasonable to try these different situations in a model of our system before actually changing the system itself. Changing the model can tell us what changes to make in the real system. A model is not a perfect representation of our real system but should be a valid approximation.

A simulation is a program or procedure that tries to model a particular event or system, such as a grocery store checkout line, a barber shop, an assembly line, a computer system or even a guided missile. A discrete event simulation is a simulation which models a system and executes a finite number of steps and operations before reaching a logical conclusion.

There are several simulation languages available such as Simula, PAWS, and GPSS. The language we would like to examine is GPSS (General Purpose Simulation System). GPSS was designed in the 1960's to aid in building models of general queuing systems. The simulations written in GPSS model real life systems, e.g., a grocery store. The modeler has available all of the facilities needed to create the events involved in a simulation of a grocery store such as entering the store, picking out items, entering a check-out line, being checked out, and leaving the store. A transaction, in this case a customer in the store, is an object that is generated at certain times in the model and then is moved through the model. The actions in the simulation are carried out on the transaction. In the case of the grocery store the transaction is the customer, and his or her actions include entering the store, shopping, entering a line, checking out, and leaving the store. In the simulation, we are only concerned with the notions that are important to modeling the grocery store itself and not the notions that would not have an influence on the results of the simulation, such as the sex of the customer. These actions are represented in GPSS by generating a transaction (customer enters the store), advancing the clock (time spent selecting items), entering a queue (entering a check-out line), seizing a facility and advancing the clock (checking out items), and finally departing the queue and terminating the transaction (leaving the store). GPSS is a language

made up of a series of procedural calls written in FORTRAN. A procedural call in GPSS is called a block and a GPSS program consists of a sequence of blocks. These procedures enable the modeler to create and use queues, facilities, and resources. The modeler also has the ability to test and assign values, simulate the passage of time, and do many other things necessary to build a model of some system. The modeler makes use of these procedures by generating one or more transactions. Some examples of transactions are a customer in a line, an automobile on an assembly line, or a process in a computer system that enters one or more blocks of the program. During the simulation, statistics are gathered about the model, such as the number of transactions generated, the number of transactions entering a queue and statistics on resources; then the simulation is terminated either by a timer or a count on the number of transactions generated during the simulation. A listing of all the statistics is printed, giving all the information about the simulation, including information about the runtime and other system considerations. It is up to the modeler to interpret these statistics and draw conclusions about the simulation based on the model. Relating statistics about things such as queues and resources to the real life entities which are being simulated, and then drawing a logical conclusion about them, is one of the hardest parts of modeling a system.

Ada was a language developed in the late 1970's and early 1980's for the Department of Defense. Many of its features make Ada a good design and software engineering language. Ada allows Modular Design, the grouping of similar items together in what are called packages. Packages are divided into two parts, the package spec and the package body. The package spec includes the procedure and function specifications, task and task type specifications (task and task types will be discussed in the next section), record declarations, and other type and object declarations. Here is an example of a package spec:

Example 1.1. An example of a package specification.

```
package test_routines is
type relational_operator is (e,ne,l,le,g,ge,min,max);
type logic_operator is (ls,lr,u,nu,i,ni,se,sne,sf,snf);
function test (operator:in relational_operator;p1,p2:in
float; transaction_name:in integer) return boolean;
task internal_test is
```

```
entry check_integer (operator:in relational_operator;
p1,p2:in integer; check_ok:in out boolean);
entry check_float (operator:in relational_operator;
p1,p2:in float; check_ok:in out boolean);
entry check_boolean (operator:in relational_operator;
p1,p2:in boolean; check_ok:in out boolean);
entry shutoff;
end internal_test;
end test_routines;
```

The package body includes all of the procedure and function bodies that coincide with their specs listed in the package specification. Also included in the package body are the task bodies for the task and task types listed in the specification. Here is an example of the package body that goes with the package spec listed in Example 1.1:

Example 1.2. An example of a package body.

```
package body test_routines is
function test (operator:in relational_operator;p1, p2:in float;
transaction_name:in integer) return boolean is
begin
    "
    "
    "
end test;
task body internal_test is
begin
main:loop
select
    accept check_float (operator:in relational_operator;
    p1,p2:in float; check_ok:in out boolean) do
        "
        "
```

```
        "
    end check_float;
or
accept check_integer (operator:in relational_operator;
p1,p2:in integer;
check_ok:in out boolean) do
    "
    "
    "
    end check_integer;
or
accept check_boolean (operator:in relational_operator;
p1,p2:in boolean; check_ok:in out boolean) do
    "
    "
    "
end check_boolean;
or
accept shutoff;
exit main;
or
terminate;
end select;
end loop main;
end internal_test;
end test_routines;
```

These specifications along with their bodies are separate compilable units and combining packages aids in creating a modular software system.

Chapter 2

The Ada Simulation Tool

SECTION I. DEVELOPMENT OF A SIMULATION TOOL.

This tool is not an attempt to write a GPSS compiler in Ada but to write a tool in Ada, to be used in conjunction with Ada for creating discrete event simulations. GPSS is used as a basis for creating this tool. Our aim is not to describe the way to write simulations in Ada but to write GPSS style simulations in Ada. A comparison is then made in chapter four to see which language was more effective in such areas as ease of creating a model, flow of control, run-time and flexibility of the language. This simulation tool known as Simulation Ada (Sim_Ada) is meant to be used with the Ada language for creating simulations. These procedures and functions are similar to those in GPSS in both name and operation, but the technique and style in which these operations were written in Sim_Ada are quite different from those written in GPSS.

The strategy for using this tool is quite different from that of GPSS. The differences will be explained in the next section, but first it would be better to explain the use and results of the procedures and functions contained in the Sim_Ada tool. Below is a list of the procedures and functions available to the user which follow the same naming scheme as GPSS.

Example 2.1. A list of procedures and functions available to the user follows. The complete specifications appear in Appendix III.

1. PROCEDURE priority (class, transaction_name:IN INTEGER);
   This procedure allows a transaction to be assigned an integer value given by "class"; initially the transaction's priority is zero. Assigning a transaction a higher priority gives that transaction certain advantages over a transaction of lower priority. Other advantages will be explained in the descriptions of the other procedures and functions.
2. PROCEDURE start (transaction_name: OUT INTEGER);
   This procedure gives each transaction task (transaction task will be explained in the

next section) its name and allows that transaction to begin the simulation. Each transaction task must start with this procedure call before invoking another Sim_Ada procedure or function call.

3. PROCEDURE terminate_transaction (transaction_name:IN INTEGER);
   This procedure is used at the end of each transaction task and takes the transaction named by "transaction_name" out of the simulation, and recycles it by resetting all of its parameters and replacing it on the list of available transactions.
4. PROCEDURE advance (median,width,transaction_name:IN INTEGER);
   This procedure simulates the passage of time. The time is calculated by taking the current clock time and then randomly generating a time. The transaction, in the advance procedure, cannot move further in the simulation until this calculated time matches the current clock time. This procedure allows the passage of time for events such as checking-out items in a grocery store, or a barber cutting a customer's hair.
5. FUNCTION running RETURN BOOLEAN;
   This function is used in the main task, which will be explained in the next section, to keep track of the time and to exit the main loop at the end of the simulation. This function does not have a counterpart in the GPSS language, but is necessary when using the Sim_Ada tool.
6. PROCEDURE queue (queue_name,transaction_name: IN INTEGER);
   This procedure allows a transaction to enter a queue given by "queue_name". The type of queue used is a priority queue with FIFO, i.e., "first in first out" in each priority class. For example, if a transaction with a priority class of zero is already in the queue, and a transaction with a priority class of one or higher enters the same queue, then this transaction will leave this queue, before the transaction with priority, class of zero. The reason for this is that transactions with the same priority class depart the queue in a FIFO manner.
7. PROCEDURE depart (queue_name,transaction_name:IN INTEGER);
   This procedure allows a transaction given by "transaction_name" to leave the queue given by "queue_name". A transaction that enters a queue must eventually leave that queue, so the queue and depart procedures must be used as a pair.
8. PROCEDURE seize (facility_name,transaction_name:IN INTEGER);
   The procedure seize allows a transaction given by "transaction_name" to capture or hold a facility or resource. Some examples of resources are a barber in a barber shop,

a clerk in a check-out line, a teller in a bank, and a C.P.U. in a computer system. A facility given by "facility_name" is anything that a transaction can capture and hold for some specific amount of time.

9. PROCEDURE release (facility_name,transaction_name:IN INTEGER);

   Procedure release allows a transaction given by "transaction_name" the ability to release or uncapture a facility given by "facility_name". As in the queue and depart procedures, the seize and release procedures must be used in pairs.

10. PROCEDURE enter (storage_name, storage_count, transaction_name:IN INTEGER);
11. PROCEDURE leave (storage_name, storage_count, transaction_name:IN INTEGER);
12. PROCEDURE generate (median, width, no_of_trans:IN INTEGER;
    time:IN OUT INTEGER;pa:IN OUT p);

    Procedure generate allows a transaction to enter the model at a time given by "time" and then sets up the next time a transaction will enter the model by randomly generating a time equal to time plus a number between median minus the width and median plus the width. The number of transactions entering the model at one time is given by "no_of_trans," and "pa" is the name of the transaction task type.

13.a. function test (operator:IN relational_operator;p1, p2: IN FLOAT;
    transaction_name:IN INTEGER) RETURN BOOLEAN;

13.b. function test (operator:IN relational_operator;p1, p2:IN INTEGER;
    transaction_name:IN INTEGER) RETURN BOOLEAN;

13.c. function test (operator:IN relational_operator; p1, p2:IN BOOLEAN;
    transaction_name:IN INTEGER) RETURN BOOLEAN;

    Function test in all three instances allows the two variables given by "p1" and "p2" to be tested against one another and returns a value of true if the test is true, and false otherwise. The relationships being tested in 13.a and 13.b are equal, not equal, less than, less than or equal, greater than and greater than or equal. The order is given by, for example, is p1 equal to p2 or is p1 less than p2? In 13.c., the only relationships tested are equal and not equal.

14. FUNCTION select_one (operator:IN relational_operator;parameter, lower,
    upper:IN INTEGER;p1:IN p;p2:IN p_array;
    transaction_name:IN INTEGER) RETURN BOOLEAN;

    Function select_one gives a transaction given by "transaction_name," the ability to test a relationship in a range of entities bounded by "lower" and "upper". The first

entity in the array "p2" that satisfies the relationship with "p1," its index in the array is put into the variable "parameter," which is a transaction parameter. An example should help clarify what this means. If one wanted to check to see which queue was empty in a range of queues and put the number of the first queue which satisfies this condition into parameter one we would have the following call to

select_one(e,1,1,8,0,q,transaction_name).

Here "e" is the relational operator for equals, the range of queues is one to eight and the relationship is read as "is p2 equal to p1," which is opposite of the function test. If a true relationship is found in the range, then a value of TRUE is returned in the function; otherwise, a value of FALSE is returned.

15. PROCEDURE start_simulation (sim_clock:IN INTEGER);

    Procedure start_simulation is used at the beginning of a simulation and is used to do general housekeeping of tasks such as initializing entities and etc. The total time the model is to run is given by "sim_clock" and is assigned to the relative and absolute clocks. Procedure start_simulation must be used at the beginning of each simulation and is used in conjunction with procedure end_simulation which is used at the end of a simulation.

16. PROCEDURE end_simulation;

    Procedure end_simulation is used at the end of each simulation. This procedure does general housekeeping associated with the end of a simulation, such as printing out statistics, shutting off other tasks, etc. Like the queue and depart procedures, procedure end_simulation must be used along with procedure start_simulation.

17. PROCEDURE assign (parameter,value,transaction_name:IN INTEGER);

    Procedure assign gives a transaction's, given by "transaction_name," parameter, given by "parameter," an integer value, given by "value". A transaction's parameters range from 1 to 127. All of a transaction's parameters are initialized to zero and to obtain a parameter's value one must reference it in the following manner:

    p(parameter) = value, where parameter has a range from 1 to 127.

    In all of these procedures and functions, either constants or variables may be used for the parameters. This allows more flexibility in the use of Sim_Ada because one can embed Ada code such as assignment statements, loops, and decision blocks, right along with the Sim_Ada procedure and function calls.

Section II. Creation and Use of the Sim_Ada Tool.

To create a model in Ada using Sim_Ada, one must have a thorough understanding of Ada's tasking facility. There are two parts to creating a model using Sim_Ada, the main task and the transaction task types. Both parts are explained below.

The main task is the simpler part of the model. It includes the beginning and ending procedures: start_simulation and end_simulation. It usually contains a main loop, which is controlled by using the running function, exiting the loop when the given time is up, and calls to a generate transactions which routes transactions to their appropriate transaction tasks.

For each transaction task type, there is a corresponding generate in the main task. These generate procedures are instantiated from the generic package gen_routines and have the following instantiation:

Example 2.2. Instantiation of the generic package gen_routines.

```
package my_generate is new gen_routines (transaction_task_type,
transaction_task_type_pointer, transaction_task_type_pointer_array);
```

In this instantiation, transaction_task_type is of type task type, transaction_task_type_pointer is type access to transaction_task_type and transaction_task_type_pointer_array is an array of pointers to transaction_task_type.

This instantiation must be used for each different transaction task type. An example of how a main task should appear follows.

Example 2.3. Example of a main task.

```
with gen_routines;
package generate_1 is new gen_routines (p1,p2,p3);
with gen_routines;
package generate_2 is new gen_routines (q1,q2,q3);
"       "
"       "
```

```
    with gen_routines;
    package generate_N is new gen_routines (r1,r2,r3);main_task is
    -- any declarations needed.
begin
    start_simulation (time);
    while (running) loop
            generate_1.generate (m_1,w_1,n_o_t_1,t_1,t_n_1,p3);
            generate_2.generate (m_2,w_2,n_o_t_2,t_2,t_n_2,q3);
            "       "
            "       "
            generate_N.generate (m_N,w_N,n_o_t_N,t_N,t_n_N,r3);
            -- m = median, w= width, n_o_t = no_of_trans, t = time
            -- t_n = transaction_name and p3,q3,r3 are all of type
            -- transaction_task_type_pointer_array
    end loop;
    end_simulation;
main_task;
```

The transaction task types have a corresponding transaction task which contain the actions of the transaction. All of the different transaction task types and transaction tasks are grouped together and placed into one package. The transaction task is where the Sim_Ada tool is mainly used, since it is here that the transactions carry out their actions, such as entering queues, and capturing facilities.

Each transaction task is dynamically created via a call to the appropriate generate procedure. When a call is made to one of these generate procedures a transaction is taken from a pool of transactions, not currently being used, and is sent to an inactive transaction task. If transaction tasks are unavailable, then one is created dynamically. Once the transaction has carried out its actions in the simulation, then both the transaction and the transaction task are recycled and used again. An example of how a package of transaction task types should appear follows:

Example 2.4. A package of transaction task types.

```
package transaction_tasks is
    task type transaction_task_1;
    type transaction_task_pointer_1 is access transaction_task_1;
    type transaction_task_pointer_array_1 is array(1..100) of transaction_task_pointer_1;
    task type transaction_task_2;
    type transaction_task_pointer_2 is access transaction_task_2;
    type transaction_task_pointer_array_2 is array(1..100) of
    "     "
    "     "
    task type transaction_task_N;
    type transaction_task_pointer_N is access transaction_task_N;
    type transaction_task_pointer_array_N is array(1..100) of transaction_task_pointer_N;
end transaction_tasks;
package body transaction_tasks is
    task body transaction_task_1 is
    name:integer;
    begin
loop
        start (name);
        -- other Sim_Ada procedure and function calls needed to create a simulation
        terminate_transaction (name);
        if (ended) then
            exit;
    end loop;
    end transaction_task_1;
    task body transaction_task_2 is
        name:integer;
    begin
    loop
        start (name);
        -- other Sim_Ada procedure and function calls needed to create a simulation
```

```
        terminate_transaction (name);
        if (ended) then
            exit;
        end loop;
    end transaction_task_2;
    "       "
    "       "
    task body transaction_task_N is
    name:integer;
    begin
    loop
        start (name);
        -- other Sim_Ada procedure and function calls needed to create a simulation
        terminate_transaction (name);
        if (ended) then
        exit;
    end loop;
    end transaction_task_N;
end transaction_tasks;
```

In the main task with the start_simulation and end_simulation procedures, the start and terminate procedures must be used at the beginning and ending of each transaction task. The start procedure is linked to a corresponding generate procedure and passes a transaction to its proper transaction task, giving the task its name. The terminate procedure takes the transaction out of the transaction task, and recycles them to be used again when needed. The function ended is used to exit the loop and shut down the task at the end of simulation.

Chapter 3

Simulation Problems and their Solutions

All three simulations are written in two manners, one using GPSS and the other using the Sim_Ada tool. A listing of each is given in Appendix I. The three simulations are:

1. A check-out line.
2. A inventory control problem.
3. A multiuser computer system.

Example 3.1. A check-out line.

Consider the check-out procedure in a super market. After customers have selected their purchases, they then proceed to the check-out counter. Except for the busiest part of the day, not all of these counters are staffed. There are always just enough check-out counters such that a customer must wait in line before receiving service. The reason is a store policy which states that if a checker has no one to serve, the counter is closed and the checker leaves the area. When the waiting lines reach a certain size, the checker returns and opens a counter. From the customers' point of view, this policy is an annoyance since it means that they must wait in line before checking-out. From the point of view of the store management it is a desirable policy, because if a customer does not wait to check out, then for some period of time before that customer arrived at the counter the checker was not active. The management wants to make effective use of the checkers' time, either at the counter, or at some other activity.

Assume that a manager of a store which operates with this policy wishes to consider a change. The manager would like to reduce the customer's waiting and needs to know how much it would cost in terms of increased checkers' time at the counters. This manager believes that reducing customer waiting time will ultimately result in more customers. The manager also fears that any improvements in service for a short trial period may increase costs and forces a return to the old policy. The manager wishes to determine the effect of the change in the policy without changing the actual operation.

A program to be developed is to simulate the activities of the checkers, i.e., checking out customers, opening and closing counters, and is to compute the time checkers spend away

from the counters. The program will first be developed to reflect the simulation with the observed operation. The computer model can be verified with accuracy, then the program will be changed to reflect the change of policy in the checkers. The effect of this change, on customer service and checker's time, can then be determined.

The specific method used in this program is only one of several approaches that could be used. It is an example of a discrete event scheduling method in which the program executes the actions that occur when a particular event alters the status of the system takes place.

In the problem the schedule of events are:

1. customer arrives,
2. customer finishes check-out,
3. counter opens,
4. counter closes.

There are other events which occur in the process; but they occur as a result of, or at the same time, as the scheduled events. Counters are to be opened when the size of the already opened lines exceeds some specified value, and the opening occurs after some time delay. Similarly counter closings are scheduled when a line becomes empty, but the closing occurs after a time delay. If a customer arrives at the counter before the end of this time delay, the customer receives service, and the scheduled closing is canceled. At the end of the simulation, statistics regarding the checkers are printed and are listed below.

A grocery store check-out model with a five second delay before closing was run for ten hours, in one second increments and results were tabulated every hour.

A. Hour one for grocery store model using GPSS.

1. Checker one:
   a. Time spent checking customers: 2801 seconds.
   b. Total number of customers: 56.
   c. Total number of customers not having to wait for checker: 56.
   d. Total number of customers left in line: 0.
   e. Average time customer waited in line: 50 seconds.
   f. Total number of times checker closed the line: 0.
   g. Total time checker spent away from check_out line: 0 seconds.
2. Checker two:

a. Time spent checking customers: 2798 seconds.

b. Total number of customers: 56.

c. Total number of customers not having to wait for checker: 56.

d. Total number of customers left in line: 0.

e. Average time customer waited in line: 49.96.

f. Total number of times checker closed the line: 0.

g. Total time checker spent away from check_out line:0 seconds.

3. Checker three:

a. Time spent checking customers: 2783 seconds.

b. Total number of customers: 56.

c. Total number of customers not having to wait for checker: 56.

d. Total number of customers left in line: 0.

e. Average time customer waited in line: 49.769.

f. Total number of times checker closed the line:0.

g. Total time checker spent away from check_out line: 0 seconds.

4. Checker four:

a. Time spent checking customers: 2765 seconds.

b. Total number of customers: 56.

c. Total number of customers not having to wait for checker: 56.

d. Total number of customers left in line: 0.

e. Average time customer waited in line: 49.393 seconds.

f. Total number of times checker closed the line:0.

g. Total time checker spent away from check_out line: 0 seconds.

5. Checker five:

a. Time spent checking customers: 0 seconds.

b. Total number of customers: 0.

c. Total number of customers not having to wait for checker: 0.

d. Total number of customers left in line: 0.

e. Average time customer waited in line: 0 seconds.

f. Total number of times checker closed the line: 1.

g. Total time checker spent away from check_out line: 1750 seconds.

6. Checker six:
    a. Time spent checking customers: 0 seconds.
    b. Total number of customers: 0.
    c. Total number of customers not having to wait for checker: 0.
    d. Total number of customers left in line: 0.
    e. Average time customer waited in line: 0 seconds.
    f. Total number of times checker closed the line: 0.
    g. Total time checker spent away from check_out line: 0 seconds.
7. Checker seven:
    a. Time spent checking customers: 0 seconds.
    b. Total number of customers: 0.
    c. Total number of customers not having to wait for checker: 0.
    d. Total number of customers left in line: 0.
    e. Average time customer waited in line: 0 seconds.
    f. Total number of times checker closed the line: 1.
    g. Total time checker spent away from check_out line: 3600 seconds.
8. Checker eight:
    a. Time spent checking customers: 0 seconds.
    b. Total number of customers: 0.
    c. Total number of customers not having to wait for checker: 0.
    d. Total number of customers left in line: 0.
    e. Average time customer waited in line: 0 seconds.
    f. Total number of times checker closed the line: 1.
    g. Total time checker spent away from check_out line: 3600 seconds.

B. Grocery store model hour one using Sim_Ada.

1. Checker one:
    a. Time spent checking customers: 2801 seconds.
    b. Total number of customers: 57.
    c. Total number of customers not having to wait for checker: 57.
    d. Total number of customers left in line: 1.
    e. Average time customer waited in line: 49 seconds.
    f. Total number of times checker closed the line: 0.

g. Total time checker spent away from check_out line: 0 seconds.

2. Checker two:
   a. Time spent checking customers: 2800 seconds.
   b. Total number of customers: 56.
   c. Total number of customers not having to wait for checker: 56.
   d. Total number of customers left in line: 0.
   e. Average time customer waited in line: 50 seconds.
   f. Total number of times checker closed the line: 0.
   g. Total time checker spent away from check_out line: 0.
3. Checker three:
   a. Time spent checking customers: 2750 seconds.
   b. Total number of customers: 56.
   c. Total number of customers not having to wait for checker: 56.
   d. Total number of customers left in line: 1.
   e. Average time customer waited in line: 49 seconds.
   f. Total number of times checker closed the line:0.
   g. Total time checker spent away from check_out line: 0 seconds.
4. Checker four:
   a. Time spent checking customers: 2750 seconds.
   b. Total number of customers: 56.
   c. Total number of customers not having to wait for checker: 56.
   d. Total number of customers left in line:1.
   e. Average time customer waited in line: 49 seconds.
   f. Total number of times checker closed the line: 0.
   g. Total time checker spent away from check_out line: 0 seconds.
5. Checker five:
   a. Time spent checking customers:0 seconds.
   b. Total number of customers: 0.
   c. Total number of customers not having to wait for checker: 0.
   d. Total number of customers left in line: 0.
   e. Average time customer waited in line: 0 seconds.
   f. Total number of times checker closed the line: 1.
   g. Total time checker spent away from check_out line: 3600 seconds.

6. Checker six:

   q. Time spent checking customers: 0 seconds.

   b. Total number of customers: 0.

   c. Total number of customers not having to wait for checker: 0.

   d. Total number of customers left in line: 0.

   e. Average time customer waited in line: 0 seconds.

   f. Total number of times checker closed the line: 1.

   g. Total time checker spent away from check_out line: 3600 seconds.

7. Checker seven:

   a. Time spent checking customers: 0 seconds.

   b. Total number of customers: 0.

   c. Total number of customers not having to wait for checker: 0.

   d. Total number of customers left in line: 0.

   e. Average time customer waited in line: 0 seconds.

   f. Total number of times checker closed the line: 1.

   g. Total time checker spent away from check_out line: 3600 seconds.

8. Checker eight:

   a. Time spent checking customers: 0 seconds.

   b. Total number of customers: 0.

   c. Total number of customers not having to wait for checker: 0.

   d. Total number of customers left in line: 0.

   e. Average time customer waited in line: 0 seconds.

   f. Total number of times checker closed the line:1.

   g. Total time checker spent away from check_out line: 3600 seconds.

Since our intention was not to solve the grocery store check-out problem using two different methods, but to compare the Sim_Ada solution to that of a standard like GPSS, it was not necessary to run the models for the full ten hours of simulated time.

Upon comparing the two above charts, it is easy to see that the two models are matched quite evenly. Some minor differences like the total number of customers generated, which differs by one, can be attributed to such things as boundary conditions. For example, is a transaction generated at time zero or is the first transaction generated at the first inter-

arrival time? Another system consideration that can vary the outcome of the model is the algorithm used to construct the random number generator.

Example 3.2. An Inventory Control Problem

An inventory is a stock of items being used for future use or sale. In a typical operation items are removed at a rate which is a constant variant. Periodically the size of the inventory is counted and if the amount present is less than some reorder point, then an order is placed for enough items to bring the inventory up to some stock control level. There are costs associated with having items in an inventory, with placing an order, and failing to have an item in stock.

The objective for studying an inventory system is to determine the best operating rules that minimize these costs. For the system, these rules would be set to a time period for counting the inventory, the reorder point, and the stock control level.

Inventory control is a significant problem in most production and distribution operations. The models used in various situations differ in many details but are similar to our simplified model.

The operation to be simulated is a retail store which sells units of a single product from an inventory and obtains replacements from a wholesale supplier.

Assumed costs for operations.

1. Taking an inventory: $50.00.
2. Receipt of an order: $15.00.
3. Loss of a sale due to lack of inventory: $10.00.
4. Cost to carry each item: $0.03 per item per day.
5. Profit per item: $20.00.

This simulation was run for an eight hour day and the time unit used was minutes.

1. Results from inventory model run in GPSS.
   a. Total number of customers: 31.
   b. Total number of sales: 27.
   c. Total profit from sales: $540.00.
   d. Total lost sales: 4.
   e. Total lost profit from lost sales: $40.00.
   f. Total number of inventories taken: 4.

g. Total cost to take all inventories: $200.00.

h. Total number of reorders: 1.

i. Total cost to handle all reorders: $15.00.

j. Total cost for holding inventory: $00.39.

k. Total profit: $284.61.

2. Results of inventory model run in Ada.

a. Total number of customers: 33.

b. Total number of sales: 27.

c. Total profit from sales: $540.00.

d. Total lost sales: 6.

e. Total lost profit from lost sales: $60.00.

f. Total number of inventories taken: 4.

g. Total cost to take all inventories: $200.00.

h. Total number of reorders: 1.

i. Total cost to handle all reorders: $15.00.

j. Total cost for holding inventory: $0.39.

k. Total profit: $264.61.

Upon comparison of the two models, one can see that they are identical except for number of lost sales and total customers. In the GPSS model, total customers were 31 and lost sales were 4, while in the Ada model, total customers were 33 and total lost sales were 6. This difference can be attributed to a difference in the way the random number generators were designed.

Example 3.3. An example of a multiuser computer system simulation.

Consider an on-line system that provides service to two terminals that are in an interactive mode; e.g., the terminal operators sends a command, waits for a response from the computer, and after a time delay sends the next command.

The terminal operation characteristics are as follows: the length of the input command from each terminal is a random number with a uniform distribution from five to eighty characters long. After a response is received from the computer, the time delay before the next input is exponentially distributed. In eighty percent of the cases, the time it takes

to execute a command is five seconds, while in twenty percent of the cases it takes thirty seconds to execute a command.

A computer operation consists of two phases, polling and execution. At one-second intervals the computer checks each individual terminal to see if there is any input waiting to be entered. If there is, the computer reads the command at a rate of three hundred characters per second. In addition to the transmission time, polling of each terminal requires one-tenth of a second.

When an output message is completed, characters are sent to the terminal at a rate of three hundred characters per second. Distribution of the output message is given in the following manner: distribution of the reply message is assumed to be ten, twenty, thirty, one hundred and one hundred fifty characters in length, with each length having a twenty percent probability of occurring.

The simulation was run twice, once in 3600 time units and once in 18,000 time units, time units being one-tenth of a second. The results are as follows:

Results for simulation run in GPSS for 3600 time units.

1. Terminal one:
    a. Number of times polled: 24.
    b. Average polling time: 11 time units.
    c. Number of request commands: 24.
    d. Average time to process request message: 0.958 time units.
    e. Number of requests executed: 24.
    f. Average execution time: 136.375 time units.
    g. Number of reply messages sent: 23.
    h. Average time to process reply message: 1.739 time units.

2. Terminal two:
    a. Number of times polled: 24.
    b. Average polling time: 11.458 time units.
    c. Number of request commands: 24.
    d. Average time to process request message: 1.125 time units.
    e. Number of requests executed: 24.
    f. Average execution time: 136.375 time units.

g. Number of reply messages sent: 23.

h. Average time to process reply message: 1.089 time units.

Results of simulation run in GPSS for 18,000 time units.

1. Terminal one:

   a. Number of times polled: 90.

   b. Average polling time: 11 time units.

   c. Number of request commands: 90.

   d. Average time to process request message: 0.978 time units.

   e. Number of requests executed: 90.

   f. Average execution time: 186.211 time units.

   g. Number of reply messages sent: 89.

   h. Average time to process reply message: 1.831 time units.

2. Terminal two:

   a. Number of times polled: 90.

   b. Average polling time: 11.122 time units.

   c. Number of request commands: 90.

   d. Average time to process request message: 1.0 time units.

   e. Number of requests executed: 90.

   f. Average execution time: 186.378 time units.

   g. Number of reply messages sent: 89.

   h. Average time to process reply message: 1.517 time units.

Results for simulation run in Ada for 3600 time units.

1. Terminal one:

   a. Number of times polled: 12.

   b. Average polling time: 11.0 time units.

   c. Number of request commands: 12.

   d. Average time to process request message: 1.0 time units.

   e. Number of requests executed: 12.

   f. Average execution time: 285.0 time units.

   g. Number of reply messages sent: 11.

   h. Average time to process reply message: 2.0 time units.

2. Terminal two:

   a. Number of times polled: 11.

   b. Average polling time: 12 time units.

   c. Number of request commands: 11.

   d. Average time to process request message: 1.0 time units.

   e. Number of requests executed: 11.

   f. Average execution time: 283.0 time units.

   g. Number of reply messages sent: 10.

   h. Average time to process reply message: 2.0 time units.

Results of simulation run in Ada for 18,000 time units.

1. Terminal one:

   a. Number of times polled: 53.

   b. Average polling time: 11 time units.

   c. Number of request commands: 53.

   d. Average time to process request message: 1.0 time units.

   e. Number of requests executed: 53.

   f. Average execution time: 325.0 time units.

   g. Number of reply messages sent: 52.

   h. Average time to process reply message: 3.0 time units.

2. Terminal two:

   a. Number of times polled: 52.

   b. Average polling time: 11.0 time units.

   c. Number of request commands: 52.

   c. Average time to process request message: 1.0 time units.

   e. Number of requests executed: 52.

   f. Average execution time: 325.0 time units.

   g. Number of reply messages sent: 51.

   h. Average time to process reply message: 3.0 time units.

Of the three examples, the computer system simulation appears to vary the most between the GPSS and the Ada version. This difference is due to the difference in random

number generators. Since the selection of the times for the request, execution and reply times were more random, then in any of the other examples, this would lead to a greater difference in the output of the two models. For this reason the GPSS and Ada versions were run twice, once for 3600 time units and once for for 18,000 time units. For the first run of 3600 time units the average number of times a transaction entered a queue was 24 for the GPSS model and 12 for the Ada model, for the second run of 18,000 time units 90 and 58 times a transaction entered a queue respectively. This seems to indicate that as the routine of the respective models gets larger the outcomes of the models get closer.

Chapter 4

Comparisons, Contrasts and Conclusions

The topics we would like to compare and contrast in this chapter include ease of the design of simulations, readability of code, use of a rigidly structured language like GPSS versus a more flexible language like Ada, and comparison of how Ada achieves parallelism versus how GPSS achieves it.

When discussing ease of design, we need to talk about two different areas. The first is the ease of design of a simulation from scratch and the second is the ease of transferring a simulation already written in GPSS to one written in Ada using Sim_Ada .

Designing a simulation in GPSS is quite different from designing a simulation in Ada with Sim_Ada. When designing a simulation in GPSS, one breaks the problem into sections according to how transactions are generated and the role they play in the simulation. However, GPSS does not allow the use of procedures, functions, or sub-routines, forcing the designer of a simulation to write a simulation as one large procedure. This presents two problems. The first is readability, which will be addressed later. The second problem is modularity and reuse of procedures, functions and modules. Since simulations in GPSS are written as one large program, it is very difficult to pick out the particular sections a designer may want to reuse in another simulation, or to examine some section of code to make some minor changes without reading through the entire program.

In the case where Ada is used to design a simulation from scratch, the designer is still faced with the same problem of splitting the simulation into manageable parts, according to how transactions are generated, and the role they play in the simulation. In this instance we can create procedures, functions, tasks, and packages in order to group common entities and routines together. This eliminates creating one large program, and breaks the problem into smaller and more manageable problems. These more manageable modules are readily reusable; for example, when creating the three examples in chapter three, we used the same main routines for all three and simply changed the instantiation of the generate routines, and changed some of the parameters of some of the procedure in the Sim_Ada tool. This strongly indicates that Ada, along with some simulation tool like Sim_Ada, is much better than GPSS, as far as ease of design is concerned.

One can transfer a GPSS program into a system of procedures, functions, tasks, and packages without great difficulty. In fact, all three examples in chapter three were first written in GPSS, and then transferred into Ada using the Sim_Ada. The designer of a simulation in Ada need only study the GPSS code and assess how and where transactions are generated, and what actions are carried out. All of the generated procedures are grouped together in the main procedure and called at the appropriate times. The actions they carry out are placed into separate tasks, and these tasks can be bound together by being located in the same package, along with any procedures and functions needed for the simulation. This makes transferring a GPSS program into a system using Ada and a tool such as Sim_Ada very systematic. The same is true for transferring a simulation written in Ada, along with a simulation tool like Sim_Ada, into a GPSS program.

When discussing readability of code, one must look at the structure of the simulation, as well as the structure of the code. Now GPSS does not allow separate procedures or functions and the designer is forced to write a simulation as one large program. This is fine for simulations of fewer than one hundred lines, but as simulations become quite large, it becomes quite cumbersome. GPSS uses what is termed "spaghetti code" constructs; for example, transfer blocks and transfer locations. A transfer block is very similar to FORTRAN's GOTO statement. A transfer location is available in such GPSS statements as the SELECT and the TEST where if these statements prove to be false, an alternative location is given for the transaction to execute. This makes the flow of control of a program very confusing and the program difficult to read.

Ada is a language designed to be readable. Its modular design, e.g., packages, allows one to write short procedures, functions, and tasks, that can be grouped into packages. Since the procedures, functions, and tasks can be written in a manner that keeps the amount of code in a smaller and more understandable size, it makes the code more readable.

GPSS is a rigidly structured language. To write a program in GPSS, the designer must adhere to a strict formatting rules, such as the name of a function must start in column eight, and the parameters must start in column twenty-one. Other formatting rules include where a comment may start, and the use of white spaces between parameters or function specifications. Other rules include variable, function or label name length and the use of capital words for all its functions and variables names.

Ada is a much more flexible language when it comes to formatting rules. There are no rules for where a line of code can start or end, whether certain words must be capitalized; indeed, Ada is not case sensitive. Ada allows the use of very long declarations. With such a capability as this, the designer can create very descriptive entities which in turn makes the code more understandable and readable.

In GPSS, the designer of a simulation is restricted to making calls to pre-defined functions available in GPSS, limited to creating functions that only return an integer value, and creating variables that are integer, real or boolean. On the other hand, a designer using Ada, along with some simulation tool, can create a simulation using the procedures, functions, or tasks included in the tool, and embed these procedure, function and task calls into an Ada program. The designer can also create procedures, functions, and tasks that return values and objects of the designer's choice.

In GPSS, parallelism is achieved through the use of current and future event chains. A transaction is placed on one of these chains along with the block name on which that transaction is currently residing. All of the transactions on the current event chain become active, when they can execute the block in which they are residing. In this manner, at any one time interval, a pseudo-parallelism is achieved.

Ada achieves parallelism in a much more natural manner through the use of tasks. Since each task is running independently of the others, and also concurrently with the others, all the actions of one transaction can be bundled into one task. This task can run with other tasks that contain the actions of other transactions, thus achieving parallelism. In this manner, there is no need to use current and future event chains, and the mechanics of building these chains is eliminated.

Other considerations when building a simulation system are areas such as compile time and run time. The advantage of Ada over GPSS when considering compile time is that a GPSS program must be recompiled every time the program is run. This becomes time consuming when it is necessary to frequently recompile the program. One of the advantages of Ada is that once a module (package) is running correctly and the designer wants to use that package in a simulation, there is no need to recompile that package. The only time one has to recompile a package is when there is a change to the package or to the entire program.

When considering run-time, GPSS clearly out-performed Ada using the Sim_Ada tool. Several factors can account for this. First, GPSS has existed since the sixties and much

optimizing has been done to the GPSS compiler in order to improve the run-time efficiency. Also GPSS was written in FORTRAN which is one of the most optimized languages written.

On the other hand, the Sim_Ada tool, which was written in Ada, was only intended of be a prototype, to prove Ada can be a good simulation language. Since extensive optimizing of this tool has not been done, any simulation written in Ada using Sim_Ada did not run as fast as the same simulation written in GPSS. Another consideration is that Ada is still a relatively new language developed in the early eighties. A language this new is not likely to be optimized as much as a language that has been around as long as FORTRAN.

In conclusion, I have shown that Ada can be used to create discrete event simulations without a great amount of difficulty provided the features of Ada are used to their fullest. The best approach to writing these simulations is the approach we choose. That is to create a tool to be used along with Ada that can create simulations and models without much difficulty and embed this tool inside a system written in Ada.

References

[1] American National Standards, "Reference Manual for the Ada Programming Language," 1983.

[2] Payne, James, "Introduction to Simulation: Programming Techniques and Methods of Analysis," McGraw-Hill, New York, 1982.

[3] Shore, R. W., Discrete-Event Simulation in Ada: Concepts, Ada Letters, Vol. 7, no. 5, pp. 105-112, October, 1987.

Appendix I

Appendix of programs/output listings for examples in chapter 3.

Example 3.1 a grocery store check-out line GPSS Program

```
* a simulation to model a grocery store check-out line
        SIMULATE
 ARIVE  FUNCTION    RN1,D16
* FUNCTION ARIVE, FUNCTION FOR INTERARRIVAL TIME FOR CUSTOMERS TO CHECK-OUT
 COT    FUNCTION    RN1,D8
0,20/.2,30/.3,50/.4,60/.6,70/.8,80/.95,90/1,100
* FUNCTION COT (CHECK OUT TIME)
        INITIAL     XH1,0/XH2,0/XH3,0/XH4,0/XH5,0/XH6,0/XH7,100/XH8,100
*XH VALUES REPRESENT COUNT ON EACH LINE, INITIALY COUNTERS 1-6 ARE OPEN
* A COUNTER IS CONSIDERED CLOSED IF IT'S XH VALUE IS 100, SO COUNTERS 7 AND 8
* ARE INITIALLY CLOSED.
*
*
        GENERATE    16 CUSTOMERS ENTER CHECK-OUT AREA
        PRIORITY    3
* CUSTOMER SELECTS SHORTEST LINE
        SELECT MIN  2,1,8,,XH
        SAVEVALUE   P2+,1,XH
        ASSIGN      3,P2
        ASSIGN      3+,8
* QUEUES 9-16 CORRESPOND TO QUEUES 1-8 BUT ARE USED FOR STATS GATHERED ABOUT A
* CUSTOMER IN LINE BEFORE CHECKING-OUT
        QUEUE       P2 customer enters selected line
        QUEUE       P3
        SEIZE       P2 customer begins checking-out
        DEPART      P3
        ADVANCE     50  TIME CHECKER USES TO CHECK ITEMS
        RELEASE     P2 checker is done checking items
        DEPART      P2 CUSTOMER LEAVES THE LINE
        SAVEVALUE   P2-,1,XH
        TERMINATE
*****************************************************************************************************
* CHECK_OUT CLERK EVENTS
        GENERATE    ,,,1
         SEIZE      7
         SEIZE      8
         SEIZE      15
         SEIZE      16
         PRIORITY   2
CLOSE    SELECT E   1,1,8,0,XH,OPEN FIND A CHECKER THAT IS IDEL AND CLOSE THAT
* LINE, IF ALL LINES ARE BUSY THEN GO TO OPEN
DELAY   ADVANCE     5 CHECKER WAITS FIVE SECONDS BEFORE STARTING TO CLOSE
        ASSIGN      2,QC(P1)
        ADVANCE     180 CHECKER WAITS THREE MINUTES TO CLOSE, IF NO CUSTOMER
* SHOWS THEN CHECKER CLOSES LINE AND GOES ON TO DO SOMETHING ELSE
        TEST E      QC(P1),P2,OPEN
        SAVEVALUE   P1,100,XH  LINE IS NOW CLOSED, A CLOSED LINE HAS A XH
```

```
* VALUE OF ONE HUNDRED
        ASSIGN      3,P1
        ASSIGN      3+,8
* FACILITIES 9-16 MAP TO FACILITIES 1-8 AND ARE USED TO GATHER STATS ABOUT
* CHECKERS 1-8 WHEN THEY ARE CLOSED AND DOING SOMETHING ELSE
        SEIZE       P1 THIS CHECKER IS NOW BUSY DOING SOMETHING ELSE
        SEIZE       P3
        TRANSFER    ,CLOSE
OPEN    SELECT MIN    1,1,8,,XH CHECK TO SEE IF ALL CHECKERS ARE BUSY OR
* CLOSED, IF THIS TRUE THEN GO TO WORK ELSE GO TO CLOSE
        TEST L      XH(P1),3,WORK
        TEST NE     XH(P1),0,CLOSE
        TEST NE     XH(P1),1
        TEST NE     XH(P1),2
        TRANSFER    ,CLOSE
WORK    SELECT E    1,1,8,100,XH,ALL3  IF ALL LINES ARE BUSY THEN GO BACK
* TO CLOSE
        ADVANCE     120  CHECKER TAKES TWO MINUTES TO OPEN
        SAVEVALUE   P1,0,XH
        ASSIGN      3,P1
        ASSIGN      3+,8
        RELEASE     P3
        RELEASE     P1  LINE IS NOW OPENE
        TEST NE     XH(P1),0
        TRANSFER    ,CLOSE
ALL3    SELECT E    1,1,8,3,XH
        TEST NE     XH(P1),3
        TRANSFER    ,CLOSE
*
*TIMER, SET IN IN SECONDS FOR TEN HOURS
*
* HOUR ONE
        GENERATE    3600
        TERMINATE   1
        START       1
        END
```

Simulation begins.

RELATIVE CLOCK: 3600.0000 ABSOLUTE CLOCK: 3600.0000

FACILITY	TOTAL TIME	AVAIL TIME	UNAVL TIME	ENTRIES	AVERAGE TIME/XACT	CURRENT STATUS	PERCENT AVAIL	SEIZING XACT	PREEMPTING XACT
1	0.778			56	50.000	AVAIL			
2	0.777			56	49.964	AVAIL		224	
3	0.773			56	49.679	AVAIL		225	
4	0.768			56	49.393	AVAIL		226	
5	0.486			1	1750.000	AVAIL		2	
7	1.000			1	3600.000	AVAIL		2	
8	1.000			1	3600.000	AVAIL		2	
13	0.486			1	1750.000	AVAIL		2	
15	1.000			1	3600.000	AVAIL		2	
16	1.000			1	3600.000	AVAIL		2	

QUEUE	MAXIMUM CONTENTS	AVERAGE CONTENTS	TOTAL ENTRIES	ZERO ENTRIES	PERCENT ZEROS	AVERAGE TIME/UNIT	$AVERAGE TIME/UNIT	QTABLE NUMBER	CURRENT CONTENTS
1	1	0.778	56	0	--	50.000	50.000		1
2	1	0.777	56	0	--	49.964	49.964		1
3	1	0.773	56	0	--	49.679	49.679		1
4	1	0.768	56	0	--	49.393	49.393		1
9	1	0.000	56	56	100.0	0.000	0.000		0
10	1	0.000	56	56	100.0	0.000	0.000		0
11	1	0.000	56	56	100.0	0.000	0.000		0
12	1	0.000	56	56	100.0	0.000	0.000		0

NON-ZERO HALFWORD SAVEVALUES: (NAME : VALUE)

2: 1
3: 1
4: 1
5: 100
7: 100
8: 100

Sim_Ada Program

```
-- an example of a grocery store check-out line
WITH transaction_routines,shopper_routines,gen_routines;
USE transaction_routines,shopper_routines;
PACKAGE customers IS NEW gen_routines(customer,customer_pointer,
customer_pointer_array);
WITH transaction_routines,shopper_routines,gen_routines;
USE transaction_routines,shopper_routines;
PACKAGE checkers IS NEW gen_routines(checker,checker_pointer,
checker_pointer_array);
WITH customers,transaction_routines,gpss_routines,
shopper_routines,c1_routines,sna_routines,checkers;
USE customers,transaction_routines,gpss_routines,
shopper_routines,c1_routines,sna_routines,checkers;
PROCEDURE grocery IS
    name,no_of_transactions: INTEGER:= 1;
BEGIN
    start_simulation(3600);
-- timer in seconds for one hour
    timer:= -1;
    checkers.generate(0,0,1,timer,checker_tasks);
    timer:= 0;
    WHILE(running) LOOP
        customers.generate(16,0,no_of_transactions,timer,
        customer_tasks);
    END LOOP;
    end_simulation;
END grocery;

WITH transaction_routines;
USE transaction_routines;
PACKAGE shopper_routines IS
    TASK TYPE customer;
    TYPE customer_pointer IS ACCESS customer;
    TYPE customer_pointer_array IS ARRAY(index_type) OF customer_pointer;
    customer_tasks: customer_pointer_array;
    TASK TYPE checker;
    TYPE checker_pointer IS ACCESS checker;
    TYPE checker_pointer_array IS ARRAY(index_type) OF checker_pointer;
    checker_tasks: checker_pointer_array;
    TASK initialize_xh;
    FUNCTION arrive RETURN INTEGER;
    FUNCTION cot RETURN INTEGER;
END shopper_routines;
```

```
WITH sna_routines,select_routines;
USE sna_routines;
PACKAGE shop_select IS NEW select_routines(INTEGER,integer_stats_array);
WITH sna_routines,utility_routines,transaction_routines,que_routines,
facility_routines,test_routines,c1_routines,shop_select;
USE sna_routines,utility_routines,transaction_routines,que_routines,
facility_routines,test_routines,c1_routines,shop_select;
PACKAGE BODY shopper_routines IS

    TASK BODY customer IS
        name:INTEGER;
    BEGIN
        transaction_routines.start(name);
        WHILE(NOT ended) LOOP
            priority(3,name);
            t(name).p(1) := cot;
            IF( shop_select.select_one(min,2,1,8,0,sna.xh,name)) THEN
-- customer picks shortest line
                t(name).p(3) := t(name).p(2) + 8;
                queue(t(name).p(2),name);
-- customer enters line
                queue(t(name).p(3),name);
-- queue for gathering stats about entering line
                sna.xh(t(name).p(2)):= sna.xh(t(name).p(2))+1;
                seize(t(name).p(2),name);
-- customer captures clerk and begins checking-out
                depart(t(name).p(3),name);
                advance(50,0,name);
-- time it takes to check-out
                release(t(name).p(2),name);
                depart(t(name).p(2),name);
-- customer leaves the line
                sna.xh(t(name).p(2)):= sna.xh(t(name).p(2))-1;
            END IF;
            terminate_transaction(name);
        END LOOP;
    END customer;

    TASK BODY checker IS
        name:INTEGER;
    BEGIN
        transaction_routines.start(name);
            seize(7,name);
            seize(8,name);
            seize(15,name);
            seize(16,name);
-- lines 7 and 8 are initially closed and facilities 15 and 16 are used for
-- gathering stats about facilities 7 and 8
            WHILE(NOT ended) LOOP
                IF(shop_select.select_one(e,1,1,8,0,sna.xh,name)) THEN
-- select the line that has no customers and begin closing the line
                    advance(5,0,name);
```

```
-- wait 5 seconds before starting to close
                    t(name).p(2) := sna.xh(t(name).p(1));
                    advance(180,0,name);
-- it takes 3 minutes to close the line
                  IF(test(e,t(name).p(2),sna.xh(t(name).p(1)),name)) THEN
-- if no customers have entered the line to check-out then after the 3 minutes
-- the line is closed
                    sna.xh(t(name).p(1)) := 100;
                    t(name).p(3) := t(name).p(1) + 8;
                    seize(t(name).p(1),name);
-- checker leaves registar and attends to something else
                    seize(t(name).p(3),name);
               END IF;
                  END IF;
                  IF(shop_select.select_one(min,1,1,8,0,sna.xh,name)) THEN
                    IF(test(ge,sna.xh(t(name).p(1)),3,name)) THEN
-- if the minimum of all the open lines is greater than or equal to 3 then
-- one of the closed lines must be opened
                          IF(shop_select.select_one(e,1,1,8,100,sna.xh,name)
-- select one of the closed lines to be opened
                              advance(120,0,name);
-- it takes 2 minutes to open a line
                              sna.xh(t(name).p(1)) := 0;
                              t(name).p(3) := t(name).p(1) + 8;
                              release(t(name).p(1),name);
                              release(t(name).p(3),name);
-- line is now open
                          END IF;
                    END IF;
               END IF;
               IF(test(ne,sna.xh(t(name).p(1)),0,name)) THEN
                  WHILE(test(e,sna.xh(t(name).p(1)),1,name)) LOOP
                    DELAY(0.1);
               END LOOP;
                  WHILE(test(e,sna.xh(t(name).p(1)),2,name)) LOOP
                    DELAY(0.1);
                  END LOOP;
               END IF;
          END LOOP;
     END checker;

     TASK BODY initialize_xh IS
     BEGIN
          FOR i IN 1 .. 6 LOOP
               sna.xh(i) := 0;
          END LOOP;
          sna.xh(7) := 100;
          sna.xh(8) := 100;
     END initialize_xh;
-- function for interarrival times for customers
     FUNCTION arrive RETURN INTEGER IS
          i,j:INTEGER;
```

```
BEGIN
     i:= (my_random_integer MOD 101);
     IF(i >= 0) AND(i<18) THEN
          j:=2;
     ELSIF(i>=18) AND (i< 33) THEN
          j:= 4;
     ELSIF(i >=33)AND(i < 45) THEN
          j:=6;
     ELSIF(i>=45) AND(i<56)THEN
          j:=8;
     ELSIF(i>=56) AND(i<64) THEN
          j:=10;
     ELSIF(i>=64) AND(i<71) THEN
          j:=12;
     ELSIF(i>=71)AND(i<77) THEN
          j:=14;
     ELSIF(i>=77)AND(i<82) THEN
          j:=16;
     ELSIF(i>=82)AND(i<86)THEN
          j:=18;
     ELSIF(i>=86)AND(i<89)THEN
          j:=20;
     ELSIF(i>=89)AND(i<92) THEN
          j:=22;
     ELSIF(i>=92)AND(i<94) THEN
          j:=24;
     ELSIF(i>=94)AND(i<96) THEN
          j:=26;
     ELSIF(i>=96)AND(i<98) THEN
          j:=28;
     ELSIF(i>=98)AND(i<99) THEN
          j:=30;
     ELSIF(i>=99)AND(i<=100) THEN
          j:=32;
     END IF;
     RETURN j;
END arrive;
-- function for determining check-out timefunction cot return integer is
     i:INTEGER;
BEGIN
     i := 10*(( my_random_integer MOD 9)+2);
     RETURN i;
END cot;
END shopper_routines;
```

Sim_Ada Listing

OUTPUT LISTING

Clock Statistics

Absolute Clock : 3601
Relative clock : 3601

Transaction Statistics

Total	Current
227	5

Queue Statistics

Name	Total	Current	Max	Zero	total time	Average residence	average zero
1	57	1	1	0	2800	49.000	49.000
2	56	0	1	0	2800	50.000	50.000
3	56	1	1	0	2750	49.000	49.000
4	56	1	1	0	2750	49.000	49.000
9	57	0	1	57	0	0.000	0.000
10	56	0	1	56	0	0.000	0.000
11	56	0	1	56	0	0.000	0.000
12	56	0	1	56	0	0.000	0.000

Facility Statistics

Name	Captured	Total	utilize	total time
1	TRUE	57	0.000	2800
2	FALSE	56	0.000	2800
3	TRUE	56	0.000	2750
4	TRUE	56	0.000	2750
5	TRUE	1	0.000	0
6	TRUE	1	0.000	0
7	TRUE	1	0.000	0
8	TRUE	1	0.000	0
13	TRUE	1	0.000	0
14	TRUE	1	0.000	0
15	TRUE	1	0.000	0
16	TRUE	1	0.000	0

non zero halfword savevalue

name	value
1	1
3	1
4	1
5	100
6	100
7	100

Example 3.2 An inventory control problem.

GPSS Program

```
* A MODEL OF AN INVENTORY CONTROL SYSTEM FOR A RETAIL STORE.
      SIMULATE
      INITIAL     XH1,20  STOCK LEVEL AT BEGINNING OF THE DAY .
*
* MODEL1
      GENERATE    15  CUSTOMERS ARRIVE
      QUEUE       1  QUEUE TO COUNT TOTAL NUMBER OF CUSTOMERS.
      TEST NE     XH1,0,LOST  IF STOCK IS DEPLETED THEN GO TO LOST ELSE CONTINUE.
      QUEUE       2  QUEUE TO COUNT TOTAL SALES.
      SAVEVALUE   1-,1,XH  REDUCE INVENTORY BY ONE.
      DEPART      2
      DEPART      1
      TERMINATE
LOST  QUEUE       3  QUEUE TO COUNT TOTAL LOST SALES.
      ADVANCE     0
      DEPART      3
      DEPART      1
      TERMINATE
*
* MODEL 2
      GENERATE    ,,,1  GENERATE ONE CONTROLLER TO HANGLE TAKING INVENTORIES
* AND MAKING REORDERS.
BACK  QUEUE       4  QUEUE TO COUNT HOW MANY INVENTORIES WERE TAKEN.
      ADVANCE     120
      DEPART      4
      TEST E      XH1,0,BACK
      QUEUE       5  QUEUE TO COUNT HOW MANY REORDERS WERE TAKEN
      ADVANCE     10  TIME IT TAKES FOR WAREHOUSE TO MAKE DELIVERY
      SAVEVALUE   1,20,XH  INVENTORY IS NOW UP TO STOCK LEVEL.
      DEPART      5
      TRANSFER    ,BACK
*
* MODEL 3
* TIMER SET FOR EIGHT HOURS IN MINUTES
      GENERATE    480
      TERMINATE   1
      START       1
      END
```

Simulation begins.

RELATIVE CLOCK: 480.0000 ABSOLUTE CLOCK: 480.0000

QUEUE	MAXIMUM CONTENTS	AVERAGE CONTENTS	TOTAL ENTRIES	ZERO ENTRIES	PERCENT ZEROS	AVERAGE TIME/UNIT	$AVERAGE TIME/UNIT	QTABLE NUMBER	CURRENT CONTENTS
1	1	0.000	31	31	100.0	0.000	0.000		0
2	1	0.000	27	27	100.0	0.000	0.000		0
3	1	0.000	4	4	100.0	0.000	0.000		0
4	1	0.979	4	0		117.500	117.500		1
5	1	0.021	1	0		10.000	10.000		0

NON-ZERO HALFWORD SAVEVALUES: (NAME : VALUE)

1: 13

Sim_Ada Program

```
-- a simulation of an inventory control system
WITH text_io;
USE text_io;
PACKAGE stock_io IS NEW integer_io(INTEGER);
WITH transaction_routines,inventory_routines,gen_routines;
USE transaction_routines,inventory_routines;
PACKAGE customers IS NEW gen_routines(customer,customer_pointer,
customer_pointer_array);
WITH transaction_routines,inventory_routines,gen_routines;
USE transaction_routines,inventory_routines;
PACKAGE controllers IS NEW gen_routines(controller,controller_pointer,
controller_pointer_array);
WITH customers,transaction_routines,gpss_routines,text_io,stock_io,
inventory_routines,c1_routines,sna_routines,controllers;
USE customers,transaction_routines,gpss_routines,text_io,stock_io,
inventory_routines,c1_routines,sna_routines,controllers;
PROCEDURE stock IS
   name,no_of_transactions: INTEGER:=1;
BEGIN
   start_simulation(481);
-- simulation was run for eight hours in time units of minutes
   timer:= -1;
   controllers.generate(0,0,1,timer,controller_tasks);
   timer:= 0;
   WHILE(running) LOOP
      customers.generate(15,0,no_of_transactions,timer,
      customer_tasks);
   END LOOP;
   end_simulation;
END stock;
```

```
WITH transaction_routines;
USE transaction_routines;
PACKAGE inventory_routines IS

    TASK TYPE customer;
    TYPE customer_pointer IS ACCESS customer;
    TYPE customer_pointer_array IS ARRAY(index_type) OF customer_pointer;
    customer_tasks: customer_pointer_array;
    TASK TYPE controller;
    TYPE controller_pointer IS ACCESS controller;
    TYPE controller_pointer_array IS ARRAY(index_type) OF controller_pointer;
    controller_tasks: controller_pointer_array;
    TASK initialize_xh;

END inventory_routines;
with sna_routines,utility_routines,transaction_routines,que_routines,
facility_routines,test_routines,c1_routines;
use sna_routines,utility_routines,transaction_routines,que_routines,
facility_routines,test_routines,c1_routines;
package body inventory_routines is

    task body customer is
        name: integer;
    begin
        transaction_routines.start(name);
        while(not ended) loop
            if(test(ne,sna.xh(1),0,name)) then
-- if inventory is not depleted then customer enters check-out line and makes
-- purchase
                queue(2,name);
-- customer enters line
                sna.xh(1):= sna.xh(1)-1;
-- inventory is decreased by one
                depart(2,name);
            else
-- lost sale due to lack of inventory
                queue(3,name);
                advance(0,0,name);
                depart(3,name);
            end if;
            terminate_transaction(name);
        end loop;
    end customer;

    task body controller is
        name: integer;

    begin
        sna.xh(1):= 20;
-- initial stock level
        transaction_routines.start(name);
        while(not ended) loop
            queue(4,name);
```

```
-- queue for keeping stats about number of times inventory is taken
            advance(120,0,name);
-- time between inventories
            depart(4,name);
            if(test(e,sna.xh(1),0,name)) then
-- if inventory is depleted then a reorder is made
                queue(5,name);
-- queue for gathering stats ab out how many reorders were made
                advance(10,0,name);
-- time it takes to make a reorder from local warehouse
                sna.xh(1):= 20;
-- stock is now up to inventory level
                depart(5,name);
            end if;
        end loop;
    end controller;

    task body initialize_xh is
    begin
        sna.xh(1):= 20;
    end initialize_xh;

end inventory_routines;
```

Sim_Ada Listing

OUTPUT LISTING

Clock Statistics

Absolute Clock : 481
Relative clock : 481

Transaction Statistics

Total	Current
34	2

Queue Statistics

Name	Total	Current	Max	Zero	total time	Average residence	average zero
2	27	0	1	27	0	0.000	0.000
3	6	0	1	6	0	0.000	0.000
4	4	1	1	0	360	90.000	90.000
5	1	0	1	0	10	10.000	10.000

Facility Statistics

Name	Captured	Total	utilize	total time

non zero halfword savevalue

name	value
1	13

Example 3.3 A multi-user computer system.

GPSS Program

```
* A SIMULATION TO MODEL A MULTI-USER COMPUTER SYSTEM
      SIMULATE
 CMD1  FUNCTION     RN1,C2
0,5/1,80
* FUNCTION CMD1 IS A CONTINOUS FUNCTION THAT SELECTS THE LENGTH OF THE
* INPUT COMMAND FROM 5 TO 80 CHARACTERS
*
 EXEC  FUNCTION     RN1,D2
*  FUNCTION EXEC GIVES THE TIME REQUIRED TO EXECUTE THE COMMAND, 20% OF THE
*  TIME IT TAKES 30 SECONDS AND 80% OF THE TIME IT TAKES 5 SECONDS.
 CMD2  FUNCTION     RN1,D5
*  FUNCTION CMD2 SELECTS THE LENGTH OF THE REPLY
*
 REPLY VARIABLE     FN$CMD2/30
*  REPLY SELECTS THE LENGTH OF THE REPLY AND THEN READS 300 CHARACTERS PER SECOND
* KEEPING IN MIND THE TIME UNITS ARE IN 1/10 OF A SECOND.
*
 REQ   VARIABLE     FN$CMD1/30
*  REQ GIVES THE TIME IT TAKES TO MAKE AN INPUT COMMAND DEPENDANT ON THE LENGTH
* OF THE COMMAND. EACH COMMAND IS READ IN 300 CHARACTERS PER SECOND
*
* TERMINAL ONE
      GENERATE     ,,,1
TERM1 QUEUE        1  QUEUE USED FOR GATHERING STATS ABOUT POLING TERMINAL 1
      SEIZE        1  FACILITY FOR POLING ALL THE TERMINALS ONE AT A TIME
      ADVANCE      11  TIME IT TAKES TO POLE ONE TERMINAL IN 1/10 OF A SEC.
      RELEASE      1
      DEPART       1
      QUEUE        3  QUEUE FOR GATHERING STATS ABOUT REQUEST COMMAND FOR
* TERMINAL 1
      SEIZE        2  I/O CHANNEL FOR INPUT COMMAND
      ADVANCE      V$REQ  TIME IT TAKES TO INPUT COMMAND
      RELEASE      2
      DEPART       3
      QUEUE        5  QUEUE FOR GATHERING STATS ABOUT EXECUTION TIME
      SEIZE        3  CAPTURE THE C.P.U.
      ADVANCE      FN$EXEC    TIME IT TAKES TO EXECUTE COMMAND.
      RELEASE      3
      DEPART       5
      QUEUE        7  QUEUE FOR GATHERING STATS ABOUT OUTPUT TO TERMINAL 1
      SEIZE        4  I/O CHANNEL BACK TO TERMINALS.
      ADVANCE      V$REPLY    TIME IT TAKES TO SEND REPLY BACK TO USER
      RELEASE      4
      DEPART       7
      TRANSFER     ,TERM1
```

```
*
* TERMINAL TWO
      GENERATE    ,,,1
TERM2 QUEUE       2  QUEUE USED FOR GATHERING STATS ABOUT POLING TERMINAL 2
      SEIZE       1  FACILITY FOR POLING ALL THE TERMINALS ONE AT A TIME
      ADVANCE     11  TIME IT TAKES TO POLE ONE TERMINAL IN 1/10 OF A SEC.
      RELEASE     1
      DEPART      2
      QUEUE       4  QUEUE FOR GATHERING STATS ABOUT REQUEST COMMAND FOR
* TERMINAL 2
      SEIZE       2  I/O CHANNEL FOR INPUT COMMAND
      ADVANCE     V$REQ  TIME IT TAKES TO INPUT COMMAND
      RELEASE     2
      DEPART      4
      QUEUE       6  QUEUE FOR GATHERING STATS ABOUT EXECUTION TIME
      SEIZE       3  CAPTURE THE C.P.U.
      ADVANCE     FN$EXEC   TIME IT TAKES TO EXECUTE COMMAND.
      RELEASE     3
      DEPART      6
      QUEUE       8  QUEUE FOR GATHERING STATS ABOUT OUTPUT TO TERMINAL 2
      SEIZE       4  I/O CHANNEL BACK TO TERMINALS.
      ADVANCE     V$REPLY    TIME IT TAKES TO SEND REPLY BACK TO USER
      RELEASE     4
      DEPART      8
      TRANSFER    ,TERM2
* TIMER IN 1/10 OF A SECOND FOR ONE-HALF HOUR
      GENERATE    18000
      TERMINATE   1
      START       1
      END
```

GPSS Listing for 3600 time units

simulation begins
RELATIVE CLOCK: 3600.0000 ABSOLUTE CLOCK: 3600.0000

FACILITY	TOTAL TIME	AVAIL TIME	UNAVL TIME	ENTRIES	AVERAGE TIME/XACT	CURRENT STATUS	PERCENT AVAIL	SEIZING XACT	PREEMPTING XACT
1	0.147			48	11.000	AVAIL			
2	0.014			48	1.042	AVAIL			
3	0.997			47	76.362	AVAIL		1	
4	0.018			46	1.413	AVAIL			

QUEUE	MAXIMUM CONTENTS	AVERAGE CONTENTS	TOTAL ENTRIES	ZERO ENTRIES	PERCENT ZEROS	AVERAGE TIME/UNIT	$AVERAGE TIME/UNIT	QTABLE NUMBER	CURRENT CONTENTS
1	1	0.073	24	0		11.000	11.000		0
2	1	0.076	24	0		11.458	11.458		0
3	1	0.006	24	8	33.3	0.958	1.437		0
4	1	0.008	24	6	25.0	1.125	1.500		0
5	1	0.909	24	0		136.375	136.375		1
6	1	0.909	24	0		136.375	136.375		1
7	1	0.011	23	9	39.1	1.739	2.857		0
8	1	0.007	23	12	52.2	1.087	2.273		0

GPSS Listing for 18,000 time units

simulation begins
RELATIVE CLOCK: 18000.0000 ABSOLUTE CLOCK: 18000.0000

FACILITY	TOTAL TIME	AVAIL TIME	UNAVL TIME	ENTRIES	AVERAGE TIME/XACT	CURRENT STATUS	PERCENT AVAIL	SEIZING XACT	PREEMPTING XACT
1	0.110			180	11.000	AVAIL			
2	0.010			180	0.989	AVAIL			
3	0.999			179	100.497	AVAIL		1	
4	0.017			178	1.674	AVAIL			

QUEUE	MAXIMUM CONTENTS	AVERAGE CONTENTS	TOTAL ENTRIES	ZERO ENTRIES	PERCENT ZEROS	AVERAGE TIME/UNIT	$AVERAGE TIME/UNIT	QTABLE NUMBER	CURRENT CONTENTS
1	1	0.055	90	0		11.000	11.000		0
2	1	0.056	90	0		11.122	11.122		0
3	1	0.005	90	29	32.2	0.978	1.443		0
4	1	0.005	90	23	25.6	1.000	1.343		0
5	1	0.931	90	0		186.211	186.211		1
6	1	0.932	90	0		186.378	186.378		1
7	1	0.009	89	36	40.4	1.831	3.075		0
8	1	0.008	89	36	40.4	1.517	2.547		0

Sim_Ada Program

```
-- a simulation of a multiuser computer system.
WITH text_io;
USE text_io;
PACKAGE computer_io IS NEW integer_io(INTEGER);
WITH transaction_routines,terminal_routines,gen_routines;
USE transaction_routines,terminal_routines;
PACKAGE terminals IS NEW gen_routines(term,term_pointer,term_pointer_array);
WITH terminals,transaction_routines,gpss_routines,text_io,computer_io,
terminal_routines,c1_routines,sna_routines;
USE terminals,transaction_routines,gpss_routines,text_io,computer_io,
terminal_routines,c1_routines,sna_routines;
PROCEDURE computer IS
    name,no_of_transactions: INTEGER:= 1;
BEGIN
    start_simulation(3600);
-- simulation was run in time units of one temth of a second
    timer:= -1;
    terminals.generate(0,0,2,timer,term_tasks);
    timer:= 0;
    WHILE(running) LOOP
        DELAY 0.1;
    END LOOP;
    end_simulation;
END computer;

WITH transaction_routines;
USE transaction_routines;
PACKAGE terminal_routines IS
    TASK TYPE term;
    TYPE term_pointer IS ACCESS term;
    TYPE term_pointer_array IS ARRAY(index_type) OF term_pointer;
    term_tasks: term_pointer_array;
    FUNCTION request RETURN INTEGER;
    FUNCTION reply RETURN INTEGER;
    FUNCTION execute RETURN INTEGER;
END terminal_routines;
```

```
WITH sna_routines,utility_routines,transaction_routines,que_routines,
facility_routines,c1_routines;
USE sna_routines,utility_routines,transaction_routines,que_routines,
facility_routines,c1_routines;
PACKAGE BODY terminal_routines IS
    TASK BODY term IS
        name: INTEGER;
    BEGIN
        transaction_routines.start(name);
        WHILE(NOT ended) LOOP
            queue(name,name);
-- queue for keeping stats about poling the terminals
            seize(1,name);
-- resource for poling terminals
            advance(11,0,name);
-- it takes one and one temth of a second to pole a terminal
            release(1,name);
            depart(name,name);
            queue(name + 2,name);
-- queue for keeping stats about requests
            seize(2,name);
-- i/o channel for handling requests
            advance(request,0,name);
-- time it takes to send request message
            release(2,name);
            depart(name + 2,name);
            queue(name + 4,name);
-- queue for gathering stats about executing requests
            seize(3,name);
-- c.p.u.
            advance(execute,0,name);
-- time it takes to execute requests
            release(3,name);
            depart(name + 4,name);
            queue(name + 6,name);
-- queue for gathering stats about replys
            seize(4,name);
-- i/o channel for handling replys
            advance(reply,0,name);
-- time it takes to handle replys
            release(4,name);
            depart(name + 6,name);
        END LOOP;
    END term;
```

```
-- function to determine how long it takes to make a request
    FUNCTION request RETURN INTEGER IS
         i: INTEGER;
    BEGIN
         I:= my_random_integer MOD 76;
         i:= i + 5;
         i:= INTEGER(i / 30);
         RETURN I;
    END request;

-- function to determine how long it takes to make a reply
    FUNCTION reply RETURN INTEGER IS
         i,j: INTEGER;

    BEGIN
         i:= my_random_integer MOD 5;
         IF(i = 0) THEN
              j:= 10;
         END IF;
         IF(i = 1) THEN
              j:= 20;
         END IF;
         IF(i = 2) THEN
              j:= 30;
         END IF;
         IF(i = 3) THEN
              j:= 100;
         END IF;
         IF(i = 4) THEN
              j:= 150;
         END IF;
         j:= INTEGER(j/30);
         RETURN j;
    END reply;

-- function to determin execution time of request
    FUNCTION execute RETURN INTEGER IS
         i,j: INTEGER;
    BEGIN
         i:= my_random_integer MOD 101;
         IF(i >= 0) AND (i <= 20) THEN
              j:= 300;
         ELSE
              j:= 50;
         END IF;
         RETURN j;
    END execute;

END terminal_routines;
```

Sim_Ada Listing for 3600 time units

OUTPUT LISTING

Clock Statistics

Absolute Clock : 3600
Relative clock : 3600

Transaction Statistics

Total	Current
2	2

Queue Statistics

Name	Total	Current	Max	Zero	total time	Average residence	average zero
1	12	0	1	0	132	11.000	11.000
2	11	0	1	0	132	12.000	12.000
3	12	0	1	7	8	0.000	1.000
4	11	0	1	8	5	0.000	1.000
5	12	1	1	0	3423	285.000	285.000
6	11	1	1	0	3122	283.000	283.000
7	11	0	1	5	16	1.000	2.000
8	10	0	1	4	14	1.000	2.000

Facility Statistics

Name	Captured	Total	utilize	total time
1	FALSE	23	0.000	253
2	FALSE	23	0.000	13
3	TRUE	22	0.000	3550
4	FALSE	21	0.000	30

non zero halfword savevalue

name	value

Sim_Ada Listing for 18,00 time units

OUTPUT LISTING

Clock Statistics

Absolute Clock : 18000
Relative clock : 18000

Transaction Statistics

Total	Current
2	2

Queue Statistics

Name	Total	Current	Max	Zero	total time	Average residence	average zero
1	53	0	1	0	583	11.000	11.000
2	52	0	1	0	583	11.000	11.000
3	53	0	1	34	27	0.000	1.000
4	52	0	1	32	26	0.000	1.000
5	53	1	1	0	17231	325.000	325.000
6	52	1	1	0	16923	325.000	325.000
7	52	0	1	25	81	1.000	3.000
8	51	0	1	21	90	1.000	3.000

Facility Statistics

Name	Captured	Total	utilize	total time
1	FALSE	105	0.000	1155
2	FALSE	105	0.000	53
3	TRUE	104	0.000	17900
4	FALSE	103	0.000	171

non zero halfword savevalue

name	value

Appendix II

Listing of Ada Packages Used to Build Sim_Ada Tool.

```
WITH text_io;
USE text_io;
PACKAGE sna_routines IS

    subtype stats_range IS INTEGER RANGE 1 .. 100;

    TYPE integer_stats_array IS ARRAY(stats_range) OF INTEGER;

    TYPE float_stats_array IS ARRAY(stats_range) OF FLOAT;

    TYPE boolean_stats_array IS ARRAY(stats_range) OF BOOLEAN;

    TYPE sna_record IS RECORD
        -- clock sna's
        c1: INTEGER; -- relative_clock
        ac: INTEGER; -- absolute clock
-------------------------------------------------------------------------
        -- facility sna's
        fe: boolean_stats_array; --list to indicate if facility exists
        f: boolean_stats_array; -- facility status
        --busy = true and not busy = false
        fc: integer_stats_array; -- facility capture count
        ftt: integer_stats_array; --total time facility was in use
        fr: float_stats_array; --facility utilization
        ft: float_stats_array; --facility average holding time
-------------------------------------------------------------------------
        -- queue sna's
        qe: boolean_stats_array; --list to indicate if queue exists
        q: integer_stats_array; -- current queue count
        qa: float_stats_array; -- average queue content
        qc: integer_stats_array; -- total queue count
        qm: integer_stats_array; --max queue count
        qtt: integer_stats_array; -- total time queue is used
        qt: float_stats_array; -- average queue residence time
        qx: float_stats_array; -- average queue residence time
        --based on qz
        qz: integer_stats_array; --total zero entry count
-------------------------------------------------------------------------
        -- storage sna's
        se: boolean_stats_array; --list to indicate if storage exists
        r: integer_stats_array; --remaining capacity in storage
        s: integer_stats_array; --current capacity of storage
        sa: float_stats_array; --average storage capacity
        sc: integer_stats_array; -- total storage count
        sr: float_stats_array; -- utilization of storage
        sm: integer_stats_array; --max storage
        st: float_stats_array; -- average holding time per unit
-------------------------------------------------------------------------
```

```
        -- transaction sna's
        te: boolean_stats_array; -- list to indicate if transaction exists
        pr: integer_stats_array; -- transaction priority level
        m1: integer_stats_array; -- transaction residence time in model
        ttc: INTEGER; --total transaction count
        ctc: INTEGER; -- current transaction count
        xh: integer_stats_array; -- xh values
    END RECORD;

    sna: sna_record;

    output: FILE_TYPE;

    PROCEDURE stats;

    PROCEDURE initialize_stats;

END sna_routines;

WITH text_io;
USE text_io;
PACKAGE sna_integer_io IS NEW integer_io(INTEGER);
WITH text_io;
USE text_io;
PACKAGE sna_float_io IS NEW float_io(FLOAT);
WITH text_io,sna_integer_io,sna_float_io;
USE text_io,sna_integer_io,sna_float_io;
PACKAGE BODY sna_routines IS

    PROCEDURE initialize_stats IS
    BEGIN
        sna.c1:= -1;
        sna.ac:= 0;
        sna.ttc:= 0;
        sna.ctc:= 0;
        FOR i IN stats_range LOOP
            sna.fe(i):= FALSE;
            sna.ftt(i):= 0;
            sna.se(i):= FALSE;
            sna.te(i):= FALSE;
            sna.f(i):= FALSE;
            sna.fc(i):= 0;
            sna.fr(i):= 0.0;
            sna.ft(i):= 0.0;
            sna.qe(i):= FALSE;
            sna.q(i):= 0;
            sna.qtt(i):= 0;
            sna.qa(i):= 0.0;
            sna.qc(i):= 0;
            sna.qm(i):= 0;
            sna.qt(i):= 0.0;
            sna.qx(i):= 0.0;
            sna.qz(i):= 0;
```

```
        sna.r(i):= 0;
        sna.s(i):= 0;
        sna.sa(i):= 0.0;
        sna.sc(i):= 0;
        sna.sr(i):= 0.0;
        sna.sm(i):= 0;
        sna.st(i):= 0.0;
        sna.pr(i):= 0;
        sna.m1(i):= 0;
        sna.xh(i):= 0;
    END LOOP;
END initialize_stats;

PROCEDURE stats IS
BEGIN
    PUT(output,"                                OUTPUT LISTING");
    NEW_LINE(output);
    PUT(output,"");
    NEW_LINE(output);
    PUT(output,"                               Clock Statistics");
    NEW_LINE(output);
    PUT(output,"   Absolute Clock :");
    PUT(output,sna.ac);
    NEW_LINE(output);
    PUT(output,"   Relative clock :");
    PUT(output,sna.c1);
    NEW_LINE(output);
    PUT(output," ");
    NEW_LINE(output);
    PUT(output,"                               Transaction Statistics");
    NEW_LINE(output);
    PUT(output,"                               Total   Current");
    NEW_LINE(output);
    PUT(output,sna.ttc,35);
    PUT(output,sna.ctc,10);
    NEW_LINE(output);
    PUT(output," ");
    NEW_LINE(output);
    PUT(output,"                                Queue Statistics");
    NEW_LINE(output);
    PUT(output,"          Name      Total   Current       Max      Zero");
    PUT(output,"          Total     Average   Average");
    NEW_LINE(output);
    PUT(output,"                                                    ");
    PUT(output,"time residence              zero");
    NEW_LINE(output);
    FOR i IN stats_range LOOP
      IF(sna.qe(i)) THEN
        IF(sna.c1 /= 0) THEN
            sna.qa(i):= FLOAT(sna.qc(i) / sna.c1);
        END IF;
        IF(sna.qc(i) /= 0) THEN
            sna.qt(i):= FLOAT(sna.qtt(i) / sna.qc(i));
```

```
            END IF;
            IF(sna.qc(i) - sna.qz(i) /= 0) THEN
               sna.qx(i):= FLOAT(sna.qtt(i) / (sna.qc(i) -
               sna.qz(i)));
            END IF;
            PUT(output,i,10);
            PUT(output,sna.qc(i),10);
            PUT(output,sna.q(i),10);
            PUT(output,sna.qm(i),10);
            PUT(output,sna.qz(i),10);
            PUT(output,sna.qtt(i),10);
            PUT(output,sna.qt(i),fore=>6,aft=>3,exp=>0);
            PUT(output,sna.qx(i),fore=>6,aft=>3,exp=>0);
            NEW_LINE(output);
        END IF;
    END LOOP;
    PUT(output," ");
    NEW_LINE(output);
    PUT(output,"                                  Facility Statistics");
    NEW_LINE(output);
    PUT(output,"      Name   Captured     Total   Utilize     Total");
    NEW_LINE(output);
    PUT(output,"                                                Time");
    NEW_LINE(output);
    FOR i IN stats_range LOOP
        IF(sna.fe(i)) THEN
            IF(sna.c1 /= 0) THEN
                sna.fr(i):= FLOAT(sna.ftt(i) / sna.c1);
            END IF;
            IF(sna.fc(i) /= 0) THEN
                sna.ft(i):= FLOAT(sna.ftt(i) / sna.c1);
            END IF;
            PUT(output,i,10);
            IF(sna.f(i)) THEN
                PUT(output,"      TRUE");
            ELSE
                PUT(output,"     FALSE");
            END IF;
            PUT(output,sna.fc(i),10);
            PUT(output,sna.fr(i),fore=>6,aft=>3,exp=>0);
            PUT(output,sna.ftt(i),10);
            NEW_LINE(output);
        END IF;
    END LOOP;
    PUT(output," ");
    NEW_LINE(output);
    PUT(output,"                          non zero halfword savevalue");
    NEW_LINE(output);
    PUT(output,"                                name     value");
    NEW_LINE(output);
    FOR i IN stats_range LOOP
        IF(sna.xh(i) /= 0) THEN
```

```
                PUT(output,i,35);
                PUT(output,sna.xh(i),10);
                NEW_LINE(output);
            END IF;
        END LOOP;
    END stats;
END sna_routines;

PACKAGE utility_routines IS

    FUNCTION get_time(median,width: IN INTEGER) RETURN INTEGER;

    FUNCTION equal_times(time,relative_clock: IN INTEGER) RETURN BOOLEAN;

    TASK start IS
        ENTRY put(name: IN INTEGER);
        ENTRY get(transaction_name: OUT INTEGER);
        ENTRY shutoff;
    END start;

    TASK count IS
        ENTRY increment(count: IN OUT INTEGER);
        ENTRY decrement(count: IN OUT INTEGER);
        ENTRY shutoff;
    END count;

    FUNCTION my_random_integer RETURN INTEGER;
    seed: INTEGER:= 13;
    mult: INTEGER:= 57;
END utility_routines;

PACKAGE BODY utility_routines IS

     FUNCTION get_time(median,width: IN INTEGER) RETURN INTEGER IS
        distance: INTEGER;
        time : INTEGER;
    BEGIN
        distance:= 2 * width + 1;
        time:= my_random_integer;
        time:= time MOD distance;
        time :=time +median-width;
        RETURN  time;
     END get_time;

    FUNCTION equal_times(time,relative_clock:IN INTEGER) RETURN BOOLEAN IS
        equal:BOOLEAN;
    BEGIN
        IF(relative_clock = time) THEN
            equal:= TRUE;
        ELSE
            equal:= FALSE;
        END IF;
        RETURN equal;
```

```
    END equal_times;

    TASK BODY start IS
    BEGIN
main:   LOOP
            SELECT
                ACCEPT PUT(name:IN INTEGER) DO
                    ACCEPT GET(transaction_name:OUT INTEGER) DO
                        transaction_name:= name;
                    END GET;
                END PUT;
            OR
                ACCEPT shutoff;
                    EXIT main;
            OR
                TERMINATE;
            END SELECT;
        END LOOP main;
    END start;

    TASK BODY count IS
    BEGIN
main:   LOOP
            SELECT
                ACCEPT increment(count: IN OUT INTEGER) DO
                    count := count+1;
                END increment;
            OR
                ACCEPT decrement(count: IN OUT INTEGER) DO
                    count:= count-1;
                END decrement;
            OR
                ACCEPT shutoff;
                    EXIT main;
            OR
                TERMINATE;
            END SELECT;
        END LOOP main;
    END count;

    FUNCTION my_random_integer RETURN INTEGER IS
        number,min: INTEGER;
    BEGIN
        number:= seed*mult;
        seed:= mult;
        min:= number/100;
        mult:= number MOD 100;
        RETURN min;
    END my_random_integer;

END utility_routines;

PACKAGE transaction_routines IS
```

```
    transaction_limit : CONSTANT INTEGER:= 100;
-- limit on the number of transactions
-- running at one time.
    SUBTYPE index_type IS INTEGER RANGE 1 .. transaction_limit;
    TYPE parameter_array IS ARRAY(0..127) OF INTEGER;
--list of parameters for each transaction.
    TYPE transaction_record IS RECORD
         p: parameter_array;
         name,relative_clock: INTEGER;
         in_use: BOOLEAN:= FALSE;
-- indicates if transaction is active in model
    END RECORD;
    TYPE points_to_transaction IS ACCESS transaction_record;
    TYPE next_transaction IS ARRAY (index_type) OF points_to_transaction;
    t: next_transaction;-- array of pointers to transaction records
    advance_time: INTEGER:= 1;
    timer,start_count,simulation_time,transaction_name,
    transaction_count: INTEGER:= 0;
    ended: BOOLEAN:= FALSE;

    PROCEDURE priority(class,transaction_name: IN INTEGER);
-- procedure priority changes priority level of transaction
-- parameter1 changes level(higher number equals higher priority)
-- parameter two assigns priority level to transaction given by transaction name

    PROCEDURE start(transaction_name: OUT INTEGER);
-- procedure start is used at beginning of each transaction task
-- to give each transaction it's proper name

    PROCEDURE terminate_transaction(transaction_name: IN INTEGER);
-- procedure terminate_transaction makes transaction available again
-- after it has completed its assignment in the simulation

    PROCEDURE assign(parameter,value,transaction_name: IN INTEGER);
    FUNCTION transaction_ok(tn:IN points_to_transaction) RETURN BOOLEAN;
    PROCEDURE reset(tn: IN OUT points_to_transaction);

END transaction_routines;
```

```
WITH utility_routines,sna_routines;
USE utility_routines,sna_routines;
PACKAGE BODY transaction_routines IS

    PROCEDURE priority(class,transaction_name: IN INTEGER) IS
    BEGIN
        IF(transaction_ok(t(transaction_name))) THEN
            t(transaction_name).p(0):= class;
        ELSE
            ended:=TRUE;
        END IF;
    END priority;

    PROCEDURE start(transaction_name: OUT INTEGER) IS
    BEGIN
        IF(sna.c1 < simulation_time) THEN
            start_count:= start_count+1;
            utility_routines.start.GET(transaction_name);
            start_count:= start_count-1;
        ELSE
            ended:= TRUE;
        END IF;
    END start;

    PROCEDURE terminate_transaction(transaction_name: IN INTEGER) IS
    BEGIN
        IF(transaction_ok(t(transaction_name))) THEN
            utility_routines.count.decrement(transaction_count);
            sna.ctc := sna.ctc-1;
            reset(t(transaction_name));
        END IF;
        WHILE(NOT t(transaction_name).in_use) LOOP
            DELAY 0.1;
        END LOOP;
    END terminate_transaction;

    PROCEDURE assign(parmeter,value,transaction_name: IN INTEGER) IS
    BEGIN
        t(transaction_name).p(parmeter):= value;
    END assign;

    FUNCTION transaction_ok(tn: IN points_to_transaction)RETURN BOOLEAN IS
        ok: BOOLEAN;
    BEGIN
        ok:= FALSE;
         IF(tn.in_use)AND(sna.c1 < simulation_time) THEN
            ok:= TRUE;
        END IF;
        RETURN ok;
    END transaction_ok;
```

```
    PROCEDURE reset(tn: IN OUT points_to_transaction) IS
    BEGIN
         tn.in_use:= FALSE;
         FOR i IN 0 .. 127 LOOP
              tn.p(i):= 0;
         END LOOP;
    END reset;

END transaction_routines;

WITH transaction_routines;
USE transaction_routines;
-- package clock_routines contains all procedures,functions and tasks
-- needed to operate the clock used for each simulation
-- clocks are referenced by sna.c1 is the relative_clock
-- and sna.ac is the absolute clock
PACKAGE c1_routines IS

    PROCEDURE advance(median,width,transaction_name: IN INTEGER);
     -- procedure advance allows the passage of time to occur
     -- transactions referenced by transaction_name are held in
     -- the advance block for a time of median + or - the width
     -- times units

    FUNCTION running RETURN BOOLEAN;
    -- function running usually used in main routine to indicate
    -- if simulation is completed

    FUNCTION advance_running RETURN BOOLEAN;

    TASK TYPE advance_task IS
         ENTRY set_advance;
         ENTRY sync_time;
         ENTRY shutoff;
    END advance_task;

    TASK time_task IS
         ENTRY inc_clock;
         ENTRY check_advance(in_advance: IN OUT BOOLEAN);
         ENTRY get_advance(median,width: IN INTEGER;
         advance_name: OUT INTEGER);
         ENTRY get_clock(transaction_clock: IN OUT INTEGER);
         ENTRY shutoff;
    END time_task;

    TYPE advance_record IS RECORD
         advance_time: INTEGER;
         in_use: BOOLEAN;
         advance_transaction: advance_task;
    END RECORD;
```

```
    TYPE advance_pointer IS ACCESS advance_record;
    TYPE advance_array IS ARRAY(index_type) OF advance_pointer;
    a: advance_array;

END c1_routines;

WITH utility_routines,transaction_routines,sna_routines;
USE utility_routines,transaction_routines,sna_routines;
PACKAGE BODY c1_routines IS

    TASK BODY advance_task IS
    BEGIN
main:   LOOP
            SELECT
                ACCEPT set_advance DO
                    ACCEPT sync_time;
                END set_advance;
            OR
                ACCEPT shutoff;
                EXIT main;
            OR
                TERMINATE;
            END SELECT;
        END LOOP main;
    END advance_task;

    PROCEDURE advance(median,width,transaction_name:IN INTEGER) IS
        advance_name:INTEGER;
    BEGIN
        IF(transaction_ok(t(transaction_name))) THEN
            IF(a(transaction_name) = NULL) THEN
                a(transaction_name):= NEW advance_record;
            END IF;
            a(transaction_name).in_use:= TRUE;
            a(transaction_name).advance_time:= sna.c1 +
            get_time(median,width);
            a(transaction_name).advance_transaction.set_advance;
        ELSE
            ended:=TRUE;
        END IF;
    END advance;

    FUNCTION running RETURN BOOLEAN IS
        sync:BOOLEAN:=FALSE;
    BEGIN
        IF(sna.c1 < simulation_time) THEN
            IF(transaction_count = 0) THEN
                time_task.inc_clock;
            ELSIF(advance_running) THEN
                DELAY 0.1;
            ELSE
                DELAY 0.1;
```

```
                time_task.inc_clock;
            END IF;
            sync:= TRUE;
        END IF;
        RETURN sync;
    END running;

    FUNCTION advance_running RETURN BOOLEAN IS
        is_running: BOOLEAN:= FALSE;
    BEGIN
        time_task.check_advance(is_running);
        RETURN is_running;
    END advance_running;

    TASK BODY time_task IS
        found: BOOLEAN;
    BEGIN
main:   LOOP
            SELECT
                ACCEPT inc_clock DO
                    sna.c1:= sna.c1 + 1;
                END inc_clock;
            OR
                ACCEPT check_advance(in_advance: IN OUT BOOLEAN) DO
                    in_advance:= FALSE;
                    found := FALSE;
                    FOR i IN index_type LOOP
                        IF(a(i) /= NULL) THEN
                            IF(a(i).in_use) THEN
                                in_advance:= TRUE;
                                IF(a(i).advance_time <= sna.c1) THEN
                                  found:= TRUE;
                                  a(i).advance_transaction.sync_time;
                                  a(i).in_use:= FALSE;
                                END IF;
                            END IF;
                        END IF;
                    END LOOP;
                    IF(NOT found) AND (in_advance) THEN
                        sna.c1:= sna.c1 + 1;
                    END IF;
                END check_advance;
            OR
                ACCEPT get_clock(transaction_clock: IN OUT INTEGER) DO
                    transaction_clock:= sna.c1;
                END get_clock;
            OR
                ACCEPT shutoff;
                EXIT main;
            OR
                TERMINATE;
            END SELECT;
        END LOOP main;
    END time_task;
END c1_routines;
```

```
WITH transaction_routines;
USE transaction_routines;
-- package for operations on queues
-- queues are priority queues with fifo on each priority level
PACKAGE que_routines IS

    TASK TYPE queue_task IS
         ENTRY hold(queue_name,in_transaction_name: IN INTEGER);
         ENTRY shutoff;
    END queue_task;

    TASK TYPE depart_task IS
         ENTRY empty(queue_name: IN INTEGER);
         ENTRY release(1 .. 10)(out_transaction_name: IN INTEGER);
         ENTRY front(queue_name,transaction_name: IN INTEGER);
         ENTRY shutoff;
    END depart_task;

    TYPE queue_task_array IS ARRAY(1 .. 10) OF queue_task;
    TYPE count_array IS ARRAY(1 .. 10) OF INTEGER;

    TYPE queue_info IS RECORD
         que: next_transaction;
         ptr_to_que: INTEGER:= 1;
         priority_queue: queue_task_array;
         priority_depart: depart_task;
         queue_count: count_array;
         total_count,current_count,total_time: INTEGER:= 0;
         name: INTEGER;
    END RECORD;

    TYPE points_to_queue IS ACCESS queue_info;
    TYPE queue_array IS ARRAY(index_type) OF points_to_queue;
    q: queue_array;

    PROCEDURE queue(queue_name,transaction_name: IN INTEGER);
     -- procedure queue places transaction referenced by transaction_name
     -- into priority queue referenced by queue_name

    PROCEDURE depart(queue_name,transaction_name: IN INTEGER);
     -- procedure depart takes transaction referenced by transaction_name
     -- out of priority queue referenced by queue_name

    PROCEDURE initialize_queue(qn: IN OUT points_to_queue;
                               queue_name:IN INTEGER);

    TASK a_que IS
         ENTRY put_in_queue(queue_name,transaction_name:IN INTEGER);
         ENTRY delete(queue_name,transaction_name: IN INTEGER);
         ENTRY shutoff;
    END a_que;
END que_routines;
```

```
WITH transaction_routines,utility_routines,sna_routines;
USE transaction_routines,utility_routines,sna_routines;
PACKAGE BODY que_routines IS
    TASK BODY queue_task IS
        priority_out: INTEGER;
    BEGIN
main:   LOOP
            SELECT
                ACCEPT hold(queue_name,in_transaction_name: IN INTEGER) DO
                    priority_out:= t(in_transaction_name).p(0);
                    q(queue_name).priority_depart.release(priority_out)
                    (in_transaction_name);
                END hold;
            OR
                ACCEPT shutoff;
                EXIT main;
            OR
                TERMINATE;
            END SELECT;
        END LOOP main;
    END queue_task;

    TASK BODY depart_task IS
        empty_queue,released_transaction,priority: INTEGER;
        queued: BOOLEAN;
    BEGIN
        ACCEPT empty(queue_name: IN INTEGER) DO
            empty_queue:= queue_name;
        END empty;
main:   LOOP
            queued:= FALSE;
            FOR i IN REVERSE 1 .. 10 LOOP
                IF(q(empty_queue).queue_count(i) > 0) THEN
                    queued:= TRUE;
                    priority:=i;
                    EXIT;
                END IF;
            END LOOP;
            SELECT
                WHEN(queued) =>
                    ACCEPT release(priority)
                                (out_transaction_name: IN INTEGER) DO
                        released_transaction:= out_transaction_name;
                    END release;
                    ACCEPT front(queue_name,transaction_name: IN INTEGER);
            OR
                ACCEPT empty(queue_name: IN INTEGER);
            OR
                ACCEPT shutoff;
                EXIT main;
            OR
                TERMINATE;
```

```
            END SELECT;
        END LOOP main;
    END depart_task;

    PROCEDURE queue(queue_name,transaction_name: IN INTEGER) IS
        priority: INTEGER;
    BEGIN
        IF(q(queue_name) = NULL) THEN
            q(queue_name):= NEW queue_info;
            initialize_queue(q(queue_name),queue_name);
            sna.qe(queue_name):= TRUE;
        END IF;
        a_que.put_in_queue(queue_name,transaction_name);
        priority:= t(transaction_name).p(0);
        q(queue_name).priority_queue(priority).hold
        (queue_name,transaction_name);
    END queue;

    PROCEDURE depart(queue_name,transaction_name: IN INTEGER) IS
    BEGIN
        a_que.delete(queue_name,transaction_name);
        q(queue_name).priority_depart.front(queue_name,transaction_name);
    END depart;

    PROCEDURE initialize_queue(qn: IN OUT points_to_queue;
                               queue_name: IN INTEGER) IS
    BEGIN
        FOR j IN 1 .. 10 LOOP
            qn.queue_count(j):= 0;
        END LOOP;
        qn.ptr_to_que:= 1;
        qn.total_count:= 0;
        qn.current_count:= 0;
        qn.total_time:= 0;
        qn.name:= queue_name;
    END initialize_queue;

    TASK BODY a_que IS
        priority:INTEGER;
    BEGIN
main:   LOOP
            SELECT
                ACCEPT put_in_queue
                (queue_name,transaction_name: IN INTEGER) DO
                    q(queue_name).que(q(queue_name).ptr_to_que):=
                    t(transaction_name);
                    q(queue_name).que(q(queue_name).ptr_to_que).
                    relative_clock:=
                    sna.c1;
                    q(queue_name).ptr_to_que:=
                    q(queue_name).ptr_to_que+1;
                    priority:= t(transaction_name).p(0);
```

```
            q(queue_name).queue_count(priority):=
            q(queue_name).queue_count(priority) +1;
            IF(sna.q(queue_name) = 0) THEN
                q(queue_name).priority_depart.empty(queue_name);
            END IF;
            sna.qc(queue_name):= sna.qc(queue_name)+1;
            sna.q(queue_name):= sna.q(queue_name)+1;
                IF(sna.q(queue_name) > sna.qn(queue_name))  THEN
                    sna.qn(queue_name):= sna.q(queue_name);
                END IF;
        END put_in_queue;
    OR
        ACCEPT delete(queue_name,transaction_name: IN INTEGER) DO
            FOR i IN 1 .. q(queue_name).ptr_to_que -1 LOOP
                IF(q(queue_name).que(i).name =
                t(transaction_name).name) THEN
                    sna.qtt(queue_name):= sna.qtt(queue_name)+
                    (sna.c1 -
                    q(queue_name)..que(i).relative_clock);
                    IF(sna.c1-
                    q(queue_name).que(i).relative_clock) =
                    0) THEN
                        sna.qt(queue_name):=
                        sna.qt(queue_name)+1;
                    END IF;
                END IF;
            END LOOP;
            priority:= t(transaction_name).p(0);
            q(queue_name).queue_count(priority):=
            q(queue_name).queue_count(priority)-1;
            sna.q(queue_name):= sna.q(queue_name) -1;
            IF(q(queue_name).que(q(queue_name).ptr_to_que-1).name=
            t(transaction_name).name) THEN
                q(queue_name).ptr_to_que:=
                q(queue_name).ptr_to_que -1;
            ELSE
                FOR j IN 1 .. q(queue_name).ptr_to_que -2 LOOP
                    IF(q(queue_name).que(j).name =
                    t(transaction_name).name) THEN
                        FOR k IN j+1 ..
                        q(queue_name).ptr_to_que -1 LOOP
                            q(queue_name).que(k-1):=
                            q(queue_name).que(k);
                        END LOOP;
                        q(queue_name).ptr_to_que:=
                        q(queue_name).ptr_to_que -1;
                    END IF;
                END LOOP;
            END IF;
        END delete;
    OR
        ACCEPT shutoff;
```

```
                EXIT main;
            OR
                TERMINATE;
            END SELECT;
        END LOOP main;
    END a_que;
END que_routines;

WITH transaction_routines;
USE transaction_routines;
-- package facility_routines contains the procedures,functions and tasks
-- needed to capture and release resources ie. check_out counters,banker
-- tellers, barbers and etc.
PACKAGE facility_routines IS

    TASK TYPE seize_task IS
        ENTRY hold(facility_name,in_transaction_name: IN INTEGER);
        ENTRY shutoff;
    END seize_task;

    TASK TYPE release_task IS
        ENTRY release(facility_name,out_transaction_name: IN INTEGER);
        ENTRY front(facility_name,transaction_name: IN INTEGER);
        ENTRY shutoff;
    END release_task;

    TYPE facility_info IS RECORD
        fac: next_transaction;
        ptr_to_facility: INTEGER:= 1;
        facility_seize: seize_task;
        facility_release: release_task;
        total_count,current_count,total_time: INTEGER:= 0;
        name: INTEGER;
    END RECORD;

    TYPE points_to_facility IS ACCESS facility_info;

    TYPE facility_array IS ARRAY(index_type) OF points_to_facility;

    f: facility_array;

    PROCEDURE seize(facility_name,transaction_name: IN INTEGER);
     -- procedure seize captures a facility referenced by facility_name
     -- and is captured by transaction referenced by transaction_name
     -- one transaction can capture a facility(resource) at a time

    PROCEDURE release(facility_name,transaction_name: IN INTEGER);
     -- procedure release allows a facility referenced by facility_name to be
     -- released by a transaction referenced by transaction_name normally
     -- the transaction that captures the facility will release the facility
     -- releasing a facility makes it possible for another transaction to
     -- capture that facility
```

```
    PROCEDURE initialize_facility(fn: IN OUT points_to_facility;
                                  facility_name: IN INTEGER);

    TASK a_facility IS
        ENTRY put_in_facility(facility_name,transaction_name: IN INTEGER);
        ENTRY delete(facility_name,transaction_name: IN INTEGER);
        ENTRY shutoff;
    END a_facility;

END facility_routines;
WITH transaction_routines,utility_routines,c1_routines,sna_routines;
USE transaction_routines,utility_routines,c1_routines,sna_routines;
PACKAGE BODY facility_routines IS

    TASK BODY seize_task IS
        priority_out: INTEGER;
    BEGIN
main:   LOOP
            SELECT
                ACCEPT hold(facility_name,in_transaction_name: IN INTEGER)
                DO
                    f(facility_name).facility_release.release
                    (facility_name,in_transaction_name);
                END hold;
            OR
                ACCEPT shutoff;
                EXIT main;
            OR
                TERMINATE;
            END SELECT;
        END LOOP main;
    END seize_task;

    TASK BODY release_task IS
        released_transaction: INTEGER;
    BEGIN
main:   LOOP
            SELECT
                ACCEPT release
                (facility_name,out_transaction_name: IN INTEGER) DO
                    released_transaction:= out_transaction_name;
                    f(facility_name).total_time:= sna.c1;
                    sna.f(facility_name):= TRUE;
                    sna.fc(facility_name):= sna.fc(facility_name)+1;
                END release;
                ACCEPT front(facility_name,transaction_name: IN INTEGER) DO
                    sna.ftt(facility_name):= sna.ftt(facility_name)+
                    (sna.c1-f(facility_name).total_time);
                    sna.f(facility_name):= FALSE;
                END front;
            OR
                ACCEPT shutoff;
```

```
                EXIT main;
            OR
                TERMINATE;
            END SELECT;
        END LOOP main;
    END release_task;

    PROCEDURE seize(facility_name,transaction_name: IN INTEGER) IS
    BEGIN
        IF(f(facility_name)=null) THEN
            f(facility_name):= NEW facility_info;
            sna.fe(facility_name):= TRUE;
            initialize_facility(f(facility_name),facility_name);
        END IF;
        a_facility.put_in_facility(facility_name,transaction_name);
        f(facility_name).facility_seize.hold
        (facility_name,transaction_name);
    END seize;

    PROCEDURE release(facility_name,transaction_name: IN INTEGER) IS
    BEGIN
        a_facility.delete(facility_name,transaction_name);
        f(facility_name).facility_release.front(facility_name,
        transaction_name);
    END release;

    PROCEDURE initialize_facility(fn:IN OUT points_to_facility;
                                  facility_name: IN INTEGER) IS
    BEGIN
        fn.ptr_to_facility:= 1;
        fn.name:= facility_name;
        fn.total_count:= 0;
        fn.current_count:= 0;
        fn.total_time:= 0;
    END initialize_facility;

    TASK BODY a_facility IS
    BEGIN
main:   LOOP
            SELECT
                ACCEPT put_in_facility
                (facility_name,transaction_name: IN INTEGER) DO
                  f(facility_name).fac(f(facility_name).ptr_to_facility):=
                  t(transaction_name);
                  f(facility_name).ptr_to_facility:=
                  f(facility_name).ptr_to_facility+1;
                END put_in_facility;
            OR
                ACCEPT delete(facility_name,transaction_name: IN INTEGER)
                DO
                    IF
```

```
                    (f(facility_name).fac(f(facility_name).ptr_to_facility
                    -1).name=t(transaction_name).name) THEN
                         f(facility_name).ptr_to_facility:=
                         f(facility_name).ptr_to_facility-1;
                    ELSE
                         FOR j IN 1 .. f(facility_name).ptr_to_facility-2
                         LOOP
                              IF(f(facility_name).fac(j).name =
                              t(transaction_name).name) THEN
                                   FOR k IN j+1 ..
                                   f(facility_name).ptr_to_facility -1
LOOP
                                   f(facility_name).fac(k-1):=
                                   f(facility_name).fac(k);
                                   END LOOP;
                                   f(facility_name).ptr_to_facility:=
                                   f(facility_name).ptr_to_facility-1;
                              END IF;
                         END LOOP;
                    END IF;
               END delete;
          OR
               ACCEPT shutoff;
               EXIT main;
          OR
               TERMINATE;
          END SELECT;
     END LOOP main;
END a_facility;
END facility_routines;
WITH transaction_routines;
USE transaction_routines;
GENERIC
TYPE a IS LIMITED PRIVATE;
TYPE b IS ACCESS a;
TYPE p IS ARRAY(index_type) OF b;
-- package gen_routines is a generic package for generating
-- different types of transactions ie. customers in a check_out line,
-- customers in a bank and etc
PACKAGE gen_routines IS

    PROCEDURE generate(median,width,no_of_trans: IN INTEGER;
    time: IN OUT INTEGER;
    pa: IN OUT p);
    -- procedure generate generates one or more transactions given by
    -- no_of_transactions at time referenced by time then generates the next
    -- transaction(s) at time time + median + or - width, then sends this
    -- transaction(s) to its task referenced by pa

    TASK generate_transaction IS
         ENTRY GET(transaction_name: OUT INTEGER;pa: IN OUT p);
         ENTRY shutoff;
    END generate_transaction;
```

```
    PROCEDURE get_transaction(name: IN OUT INTEGER; pa: IN OUT p);

    PROCEDURE initialize_transaction(tn: IN OUT points_to_transaction);

    END gen_routines;

WITH transaction_routines,utility_routines,c1_routines,sna_routines;
USE transaction_routines,utility_routines,c1_routines,sna_routines;
PACKAGE BODY gen_routines IS

    PROCEDURE  generate(median,width,no_of_trans: IN INTEGER;
    time :IN OUT INTEGER;
    pa: IN OUT p) IS
         name,old_time: INTEGER;
         ended_in: BOOLEAN:=FALSE;

    BEGIN
         IF(time<=sna.c1) THEN
              old_time:= time;
              time:= get_time(median,width);
              time:= time+old_time;
              FOR i IN 1 .. no_of_trans LOOP
                   utility_routines.count.increment(transaction_count);
                   sna.ttc:= sna.ttc+1;
                   sna.ctc:= sna.ctc+1;
                   generate_transaction.GET(name,pa);
              END LOOP;
         END IF;
    END generate;

    TASK BODY generate_transaction IS
         name: INTEGER;
    BEGIN
main:    LOOP
              SELECT
                   ACCEPT GET(transaction_name: OUT INTEGER;pa: IN OUT p) DO
                        get_transaction(name,pa);
                         transaction_name:= name;
                   END GET;
              OR
                   ACCEPT shutoff;
                   EXIT main;
              OR
                   TERMINATE;
               END SELECT;
         END LOOP main;
    END generate_transaction;

    PROCEDURE get_transaction(name: IN OUT INTEGER;pa: IN OUT p) IS
         temp_name: INTEGER;
         found: BOOLEAN:=FALSE;
```

```
    BEGIN
         WHILE(NOT found) LOOP
              FOR i IN REVERSE 1 .. 50 LOOP
                   IF(t(i)/=null) THEN
                        IF(NOT t(i).in_use) THEN
                             name:= i;
                             initialize_transaction(t(name));
                             t(name).relative_clock:= sna.c1;
                             found:= TRUE;
                        END IF;
                   END IF;
              END LOOP;
              IF(NOT found) THEN
                   FOR i IN 1 .. 50 LOOP
                   IF(t(i)= null) THEN
                             pa(i):= NEW a;
                             t(i):= NEW transaction_record;
                             name:= i;
                             t(name).name:=name;
                             initialize_transaction(t(name));
                             t(name).relative_clock:= sna.c1;
                             utility_routines.start.PUT(name);
                             found := TRUE;
                             EXIT;
                        END IF;
                    END LOOP;
              END IF;
              IF(NOT found) THEN
                   DELAY 0.1;
              END IF;
         END LOOP;
    END get_transaction;

    PROCEDURE initialize_transaction(tn: IN OUT points_to_transaction)IS
    BEGIN
         tn.relative_clock:= sna.c1;
         tn.in_use:= TRUE;
         tn.p(0):= 1;
         FOR i IN 1 .. 127 LOOP
              tn.p(i):= 0;
         END LOOP;
    END initialize_transaction;

END gen_routines;

-- package tes_routines is a generic package containing the procedures,
-- functions and tasks neded to test standard numerical attributes
WITH transaction_routines;
USE transaction_routines;
PACKAGE test_routines IS

    TYPE relational_operator IS (e,ne,l,le,g,ge,min,max);
```

```
    TYPE logic_operator IS (ls,lr,u,nu,i,ni,se,sne,sf,snf);

    FUNCTION test(operator: IN relational_operator;p1,p2: IN FLOAT;
                transaction_name: IN INTEGER) RETURN BOOLEAN;

    FUNCTION test(operator: IN relational_operator;p1,p2: IN INTEGER;
                transaction_name: IN INTEGER) RETURN BOOLEAN;

    FUNCTION test(operator: IN relational_operator;p1,p2: IN BOOLEAN;
                transaction_name: IN INTEGER) RETURN BOOLEAN;

    TASK internal_test IS
         ENTRY check_integer(operator: IN relational_operator;
         p1,p2: IN INTEGER; check_ok: IN OUT BOOLEAN);
         ENTRY check_float(operator: IN relational_operator;
         p1,p2: IN FLOAT; check_ok: IN OUT BOOLEAN);
         ENTRY check_boolean(operator: IN relational_operator;
         p1,p2: IN BOOLEAN; check_ok: IN OUT BOOLEAN);
         ENTRY shutoff;
    END internal_test;

END test_routines;
WITH transaction_routines;
USE transaction_routines;
PACKAGE BODY test_routines IS

    FUNCTION test(operator: IN relational_operator;p1,p2: IN FLOAT;
                transaction_name: IN INTEGER) RETURN BOOLEAN IS
         test_ok: BOOLEAN:= FALSE;
    BEGIN
         IF(transaction_ok(t(transaction_name))) THEN
              internal_test.check_float(operator,p1,p2,test_ok);
         END IF;

         RETURN test_ok;
    END test;

    FUNCTION test(operator: IN relational_operator;p1,p2: IN INTEGER;
              transaction_name: IN INTEGER) RETURN BOOLEAN IS
         test_ok: BOOLEAN:= FALSE;
    BEGIN
         IF(transaction_ok(t(transaction_name))) THEN
              internal_test.check_integer(operator,p1,p2,test_ok);
         END IF;
         RETURN test_ok;
    END test;

    FUNCTION test(operator: IN relational_operator;p1,p2: IN BOOLEAN;
                transaction_name: IN INTEGER) RETURN BOOLEAN IS
         test_ok:BOOLEAN:= FALSE;
       IF(transaction_ok(t(transaction_name))) THEN
internal_test.check_boolean(operator,p1,p2,test_ok);
```

```
        END  IF;
        RETURN test_ok;
    END test;

    TASK BODY internal_test IS
    BEGIN
main:   LOOP
            SELECT
                ACCEPT check_float(operator: IN relational_operator;
                p1,p2: IN FLOAT; check_ok:IN OUT BOOLEAN) DO
                    CASE operator IS
                        WHEN l =>
                            IF(p1<p2)THEN
                                check_ok:= TRUE;
                            ELSE
                                check_ok:=FALSE;
                            END IF;
                        WHEN le =>
                            IF(p1<=p2)THEN
                                check_ok:=TRUE;
                            ELSE
                                check_ok:=FALSE;
                            END IF;
                        WHEN g =>
                            IF(p1>p2)THEN
                                check_ok:=TRUE;
                            ELSE
                                check_ok:=FALSE;
                            END IF;
                        WHEN ge =>
                            IF(p1>=p2)THEN
                                check_ok:=TRUE;
                            ELSE
                                check_ok:=FALSE;
                            END IF;
                        WHEN e =>
                            IF(p1=p2)THEN
                                check_ok:=TRUE;
                            ELSE
                                check_ok:=FALSE;
                            END IF;
                        WHEN ne =>
                            IF(p1/=p2)THEN
                                check_ok:=TRUE;
                            ELSE
                                check_ok:=FALSE;
                            END IF;
                        WHEN others =>
                            null;
                    END CASE;
                END check_float;
```

```
OR
     ACCEPT check_integer(operator: IN relational_operator;
     p1,p2: IN INTEGER; check_ok: IN OUT BOOLEAN) DO
          CASE operator IS
               WHEN l =>
                    IF(p1<p2)THEN
                         check_ok:=TRUE;
                    ELSE
                         check_ok:=FALSE;
                    END IF;
               WHEN le =>
                    IF(p1<=p2)THEN
                         check_ok:=TRUE;
                    ELSE
                         check_ok:=FALSE;
                    END IF;
               WHEN g =>
                    IF(p1>p2)THEN
                         check_ok:=TRUE;
                    ELSE
                         check_ok:=FALSE;
                    END IF;
               WHEN ge =>
                    IF(p1>=p2)THEN
                         check_ok:=TRUE;
                    ELSE
                         check_ok:=FALSE;
                    END IF;
               WHEN e =>
                    IF(p1=p2)THEN
                         check_ok:=TRUE;
                    ELSE
                         check_ok:=FALSE;
                    END IF;
               WHEN ne =>
                    IF(p1/=p2)THEN
                         check_ok:=TRUE;
                    ELSE
                         check_ok:=FALSE;
                    END IF;
               WHEN others =>
                    null;
          END CASE;
     END check_integer;
OR
     ACCEPT check_boolean(operator: IN relational_operator;
     p1,p2: IN BOOLEAN; check_ok: IN OUT BOOLEAN) DO
          CASE operator IS
               WHEN e =>
                    IF(p1=p2)THEN
                         check_ok:=TRUE;
                    ELSE
                         check_ok:=FALSE;
                    END IF;
```

```
                    WHEN ne =>
                        IF(p1/=p2)THEN
                            check_ok:=TRUE;
                        ELSE
                            check_ok:=FALSE;
                        END IF;
                    WHEN others =>
                        null;
                END CASE;
            END check_boolean;
        OR
            ACCEPT shutoff;
            EXIT main;
        OR
            TERMINATE;
        END SELECT;
    END LOOP main;
END internal_test;

END test_routines;

-- package select_routines is a generic package containing the procedures,
-- functions
-- and tasks neded to test standard numerical attributes

WITH transaction_routines,test_routines,sna_routines;
USE transaction_routines,test_routines,sna_routines;
GENERIC
TYPE p IS (<>);
TYPE p_array IS ARRAY (stats_range) OF p;
PACKAGE select_routines IS

    FUNCTION select_one(operator: IN relational_operator;parmeter,lower,
                upper: IN INTEGER;p1: IN p;p2: IN p_array;
                transaction_name: IN INTEGER) RETURN BOOLEAN;

END select_routines;

WITH transaction_routines,test_routines;
USE transaction_routines,test_routines;
PACKAGE BODY select_routines IS

    FUNCTION select_one(operator: IN relational_operator;parmeter,lower,
                    upper: IN INTEGER;p1: IN p;p2: IN p_array;
                    transaction_name: IN INTEGER) RETURN BOOLEAN IS
        select_ok: BOOLEAN:=FALSE;
        least,greatest: p;
    BEGIN
        IF(transaction_ok(t(transaction_name))) THEN
            CASE operator IS
                WHEN min =>
```

```
        t(transaction_name).p(parmeter):= lower;
        least := p2(lower);
        select_ok:= TRUE;
        FOR i IN lower +1 .. upper LOOP
             IF(p2(i) < least) THEN
                  least:= p2(i);
                  t(transaction_name).p(parmeter) := i;
             END IF;
        END LOOP;
   WHEN max =>
        t(transaction_name).p(parmeter):=lower;
        greatest := p2(lower);
        select_ok:= TRUE;
        FOR i IN lower +1 .. upper LOOP
        IF(p2(i) > greatest)THEN
             greatest:= p2(i);
             t(transaction_name).p(parmeter) := i;
        END IF;
   END LOOP;
   WHEN l =>
        FOR i IN lower .. upper LOOP
             IF(p1 < p2(i)) THEN
                  t(transaction_name).p(parmeter):=i;
                  select_ok:= TRUE;
                  RETURN select_ok;
                  EXIT;
             END IF;
        END LOOP;
                  WHEN le =>
        FOR i IN lower .. upper LOOP
             IF(p1 <= p2(i)) THEN
                  t(transaction_name).p(parmeter):=i;
                  select_ok:= TRUE;
                  RETURN select_ok;
                  EXIT;
             END IF;
        END LOOP;
                           WHEN g =>
        FOR i IN lower .. upper LOOP
             IF(p1 > p2(i)) THEN
                  t(transaction_name).p(parmeter):=i;
                  select_ok:= TRUE;
                  RETURN select_ok;
                  EXIT;
             END IF;
        END LOOP;
                           WHEN ge =>
        FOR i IN lower .. upper LOOP
             IF(p1 >= p2(i)) THEN
                  t(transaction_name).p(parmeter):=i;
                  select_ok:= TRUE;
                  RETURN select_ok;
                  EXIT;
```

```
                END IF;
            END LOOP;
                                WHEN e =>
            FOR i IN lower .. upper LOOP
                IF(p1 = p2(i)) THEN
                    t(transaction_name).p(parmeter):=i;
                    select_ok:= TRUE;
                    RETURN select_ok;
                    EXIT;
                END IF;
            END LOOP;
                                WHEN ne =>
            FOR i IN lower .. upper LOOP
                IF(p1 /= p2(i)) THEN
                    t(transaction_name).p(parmeter):=i;
                    select_ok:= TRUE;
                    RETURN select_ok;
                    EXIT;
                END IF;
            END LOOP;

        END CASE;
      END IF;
      RETURN select_ok;
  END select_one;

END select_routines;
```

```
WITH transaction_routines,que_routines;
USE transaction_routines,que_routines;
-- package GPSS_routines contains the procedures,functions and tasks
-- needed to set up a simulation such as initialize queues,transactions,
-- facilities and etc also routines to finish or reset simulations such as
-- clear queues free facilities and etc
PACKAGE GPSS_routines IS

    PROCEDURE start_simulation(sim_clock: IN INTEGER);
    -- procedure start_simulation initializes entities needed for a simulation
    -- ie queues, facilities, clocks, transactions and etc also the simulation
    -- time is given by sim_clock

    PROCEDURE end_simulation;
    -- procedure end_simulation prints out the statistics
    -- of a simulation and clears the entities such as queues, facilities,
    -- clock and etc

    PROCEDURE reset;
    -- procedure reset reinitializes the relative clock and prints the
    -- present state of the simulation, such as queue and facility statistics
    -- and continues with the simulation

END GPSS_routines;

WITH transaction_routines, new_t;
USE transaction_routines;
PACKAGE new_transaction IS NEW new_t(transaction_record,points_to_transaction,
next_transaction);
WITH transaction_routines,c1_routines,new_t;
USE transaction_routines,c1_routines;
PACKAGE new_advance IS NEW new_t(advance_record,advance_pointer,
advance_array);
WITH transaction_routines,facility_routines,new_t;
USE transaction_routines,facility_routines;
PACKAGE new_facility IS NEW new_t(facility_info,points_to_facility,
facility_array);
WITH transaction_routines,que_routines,new_t;
USE transaction_routines,que_routines;
PACKAGE new_queue IS NEW new_t(queue_info,points_to_queue,queue_array);
WITH text_io,new_transaction,transaction_routines,facility_routines,
utility_routines,que_routines,c1_routines,new_advance,new_queue,
new_facility,sna_routines,test_routines;
USE text_io,new_transaction,transaction_routines,facility_routines,
utility_routines,que_routines,c1_routines,new_advance,new_queue,
new_facility,sna_routines,test_routines;
PACKAGE BODY GPSS_routines IS

    PROCEDURE start_simulation(sim_clock: IN INTEGER) IS
    BEGIN
        simulation_time:= sim_clock;
        OPEN(output,out_file,"output.f");
```

```
        initialize_stats;
        new_transaction.initialize(t);
        new_advance.initialize(a);
        new_queue.initialize(q);
        new_facility.initialize(f);
    END start_simulation;

    PROCEDURE end_simulation IS
    BEGIN
        sna.ac:= sna.ac+sna.c1;
        stats;
        ended:=TRUE;
        DELAY 3.0;
        WHILE(advance_running)LOOP
            DELAY(0.1);
        END LOOP;
        DELAY(2.0);
        FOR i IN index_type LOOP
            IF(q(i) /= null) THEN
                FOR j IN 1 .. 10 LOOP
                    q(i).priority_queue(j).shutoff;
                END LOOP;
                q(i).priority_depart.shutoff;
            END IF;
        END LOOP;
        FOR i IN 1 .. start_count LOOP
            utility_routines.start.PUT(i);
            DELAY 1.0;
        END LOOP;
        FOR i IN 1 .. transaction_limit LOOP
            IF(t(i) /= null) THEN
                t(i).in_use := TRUE;
                DELAY 1.0;
            END IF;
        END LOOP;
        CLOSE(output);
    END end_simulation;

    PROCEDURE reset IS
    BEGIN
        sna.ac := sna.ac + sna.c1;
        stats;
        sna.c1 := -1;
        timer := 0;
    END reset;

END GPSS_routines;
```

Appendix III

A list of Ada packages used in the simulation tool package utility_routines is

```
   function get_time(median,width : in integer) return integer;
   function equal_times(time,relative_clock: in integer) return
   boolean;

   task start is
         entry put(name:in integer;ended_in:in boolean);
         entry get(transaction_name:out integer;ended_out:out
         boolean);
         entry shutoff;
         end start;

  task count is
         entry increment(count :in out integer);entry
         decrement(count : in out integer);
         entry shutoff;
         end count;

   function my_random_integer return integer;
      seed:integer:=13;
      mult:integer:=57;
end utility_routines;

package transaction_routines is
    transaction_limit :constant integer:=100;
     --limit on the number of transactions
     -- running at one time.
    subtype index_type is integer range 1 .. transaction_limit;
    type parameter_array is array(0..127) of integer;
     --list of parameters for each transaction.
    type transaction_record is record
          p:parameter_array;
          name,relative_clock:integer;
          in_use:boolean:=false;
           -- indicates if transaction is active in model
    end record;

    type points_to_transaction is access transaction_record;
    type next_transaction is array (index_type) of
    points_to_transaction;
    t:next_transaction;
     -- array of pointers to transaction records
    advance_time:integer:=1;
    start_count, simulation_time, transaction_name,
    transaction_count: integer:=0;
```

```
    ended:boolean:=false;
    procedure priority(class,transaction_name:in integer);
     -- procedure priority changes priority level of transaction
     -- parameter1 changes level (higher number equals higher priority)
     -- parameter two assigns priority level to transaction given by transaction name
    procedure start(transaction_name: out integer);
    -- procedure start is used at beginning of each transaction
    -- task to give each transaction it's proper name
    procedure terminate_transaction(transaction_name:in integer);
    -- procedure terminate_transaction is used to make transaction
    -- available again after it has completed its assignment
    -- in the simulation
     procedure assign(parameter, value, transaction_name:in integer);
     function transaction_ok(tn:in points_to_transaction) return boolean;
     procedure reset(tn:in out points_to_transaction);
end transaction_routines;

with transaction_routines;
use transaction_routines;
  -- package clock_routines contains all procedures,functions and
  -- tasks needed to operate the clock used for each simulation
  -- clocks are referenced by sna.c1 is the relative_clock
  -- and sna.ac is the absolute clock
package clock_routines is
     type advance_task is limited private;
     type advance_array is private;
     a:advance_array;
     procedure advance(median, width, transaction_name:in integer);
       -- procedure advance allows the passage of time to occur
       -- transactions referenced by transaction_name are held in
       -- the advance block for a time of median + or - the width
       -- times units
    function running return boolean;
      -- function running usually used in main routine to indicate
      -- if the simulation is completed
    function advance_running return boolean;
     private
     type advance_record is record advance_time:integer;
          in_use:boolean;
          advance_transaction:advance_task;
     end record;

     type advance_pointer is access advance_record;
     type advance_array is array(index_type) of advance_pointer;
          entry set_advance;
          entry sync_time;
          entry shutoff;
     end advance_task;

     task time_task is
          entry inc_clock;
```

```
        entry check_advance(in_advance:in out boolean);
        entry get_advance(median,width:in integer;
        advance_name:out integer
        entry get_clock(transaction_clock:in out integer);
        entry shutoff;
    end time_task;
end clock_routines;

with transaction_routines;
use transaction_routines;
  -- package for operations on queues
  -- queues are priority queues with fifo on each priority level
package que_routines is
    task type queue_task is
        entry hold(queue_name, in_transaction_name:in integer);
        entry shutoff;
    end queue_task;

        task type depart_task is
        entry empty(queue_name:in integer);
        entry release(1 ..10)(out_transaction_name:in integer);
        entry front(queue_name,transaction_name:in integer);
        entry shutoff;
    end depart_task;

     type queue_task_array is array(1..10) of queue_task;
     type count_array is array(1..10) of integer;
     type queue_info is record
        que:next_transaction;
        ptr_to_que:integer:=1;
        priority_queue: queue_task_array;
        priority_depart:depart_task;
        queue_count: count_array;
        total_count,current_count,total_time:integer:=0;
        name:integer;
    end record;

    type points_to_queue is access queue_info;
    type queue_array is array(index_type) of points_to_queue;
    q:queue_array;
    procedure queue(queue_name, transaction_name:in integer);
     -- procedure queue places transaction referenced by transaction_name
     -- into priority queue referenced by queue_name
     procedure depart(queue_name,transaction_name:in integer);
     -- procedure depart takes transaction referenced by transaction_name
     -- out of priority queue referenced by queue_name
     procedure initialize_queue(qn:in out points_to_queue;
     queue_name:in integer);
     task a_que is
        entry put_in_queue(queue_name, transaction_name:in
        integer);
```

```
        entry delete(queue_name,transaction_name:in integer);
        entry shutoff;
    end a_que;

end que_routines;


with transaction_routines;
use transaction_routines;
  -- package facility_routines contains the procedures, functions
  -- and tasks needed to capture and release resources ie.
  -- check_out counters, bank tellers, barbers and etc.
package facility_routines is
    task type seize_task is
        entry hold(facility_name,in_transaction_name:in integer);
        entry shutoff;
        end seize_task;
    task type release_task is
        entry release(facility_name, out_transaction_name:in
        integer);
        entry front(facility_name,transaction_name:in integer);
        entry shutoff;
    end release_task;
    type facility_info is record
        fac:next_transaction;
        ptr_to_facility:integer:=1;
        facility_seize:seize_task;
        facility_release:release_task;
        total_count,current_count,total_time:integer:=0;
        name:integer;
    end record;
    type points_to_facility is access facility_info;
    type facility_array is array(index_type) of
    points_to_facility;
    f:facility_array;
    procedure seize(facility_name,transaction_name:in integer);
      -- procedure seize captures a facility referenced by facility_name
      -- and is captured by transaction referenced by transaction_name
      -- one transaction can capture a facility(resource) at a time
    procedure release(facility_name, transaction_name:in integer);
      -- procedure release allows a facility referenced by
      -- facility_name to be released by a transaction referenced
      -- by transaction_name normally the transaction that
      -- captures the facility will release the facility
      -- releasing a facility makes it possible for another
      -- transaction to capture that facility
    procedure initialize_facility(fn:in out points_to_facility;
    facility_name:in integer);
    task a_facility is
        entry put_in_facility(facility_name, transaction_name:in integer);
        entry delete(facility_name, transaction_name:in integer);
```

```
        entry shutoff;
    end a_facility;
end facility_routines;
with transaction_routines;
use transaction_routines;
generic
type a is limited private;
type b is access a;
type p is array(index_type) of b;
  -- package gen_routines is a generic package for generating
  -- different types of transactions ie. customers in a check_out
  -- line, customers in line at banks and etc
package gen_routines is
     procedure generate(median, width, no_of_trans:in integer;
     time:in out integer;pa:in out p);
      -- procedure generate generates one or more transactions
      -- given by no_of_transactions at time referenced by time
      -- then generates the next transaction(s) at time time +
      -- median + or - width, then sends this transaction(s) to
      -- its task referenced by pa
     task generate_transaction is
        entry get(transaction_name: out integer;pa:in out p);
        entry shutoff;
     end generate_transaction;
     procedure get_transaction(name:in out integer; pa:in out p);
     procedure initialize_transaction(tn:in out points_to_transaction);
 end gen_routines;

with transaction_routines, que_routines;
use transaction_routines, que_routines;
  -- package GPSS_routines contains the procedures,functions and
  -- tasks needed to set up a simulation such as initialize
  -- queues, transactions, facilities and etc also routines to
  -- finish or reset simulations such as clear queues free facilities and etc
package GPSS_routines is
     procedure start_simulation(sim_clock:in integer);
       -- procedure start_simulation initializes entities needed
       -- for a simulation  ie queues, facilities, clocks, transactions
       -- and etc also the simulation time is given by
       -- sim_clock
     procedure end_simulation;
       -- procedure end_simulation prints out the statistics
       -- of a simulation and clears the entities such as
       -- queues, facilities, clock and etc
end GPSS_routines;

  -- package select_routines is a generic package containing the
  -- procedures, functions and tasks needed to test standard
  -- numerical attributes with transaction_routines,
  -- test_routines, sna_routines
use transaction_routines, test_routines, sna_routines;
```

```
generic
type p is (<>);
type p_array is array (stats_range) of p;
package select_routines is
     function select_one(operator:in  relational_operator;
     parameter, lower, upper:in integer;p1:in p;p2:in p_array;
     transaction_name:in integer) return boolean;
end select_routines;

  -- package tes_routines is a generic package containing the
  -- procedures, functions and tasks needed to test standard numerical attributes
with transaction_routines;
use transaction_routines;
package test_routines is
     type relational_operator is (e, ne, l, le, g, ge, min, max);
     type logic_operator is (ls, lr, u, nu, i, ni, se, sne, sf, snf);
     function test(operator:in relational_operator;p1, p2:in
     float;transaction_name:in integer) return boolean;
     function test(operator:in relational_operator;p1, p2:in
     integer; transaction_name:in integer) return boolean;
     function test(operator:in relational_operator;p1, p2:in
     boolean; transaction_name:in integer) return boolean;
     task internal_test is
          entry check_integer(operator:in relational_operator;
          p1, p2:in integer; check_ok:in out boolean);
          entry check_float(operator:in relational_operator; p1,
          p2:in float; check_ok:in out boolean);
          entry check_boolean(operator:in relational_operator;
          p1, p2:in boolean; check_ok:in out boolean);
      entry shutoff;
    end internal_test;
end test_routines;

package sna_routines is
     subtype stats_range is integer range 1 .. 1000;
     type integer_stats_array is array(stats_range)of integer;
     type float_stats_array is array(stats_range) of float;
     type boolean_stats_array is array(stats_range)of boolean;
     type sna_record is record
          -- clock sna's c1:integer;
          -- relative_clock ac:integer;
          -- absolute clock
          -- facility sna's
          fe:boolean_stats_array;
            --list to indicate if facility exists
          f:boolean_stats_array;
            -- facility status
            -- busy = true and not busy = false
          fc:integer_stats_array;
            -- facility capture count
          ftt:integer_stats_array;
            -- total time facility was in use
```

```
    fr:float_stats_array;
      -- facility utilization
    ft:float_stats_array;
      -- facility average holding time
      -- queue sna's
    qe:boolean_stats_array;
      -- list to indicate if queue exists
    q:integer_stats_array;
      -- current queue count
    qa:float_stats_array;
      -- average queue content
    qc:integer_stats_array;
      -- total queue count
    qm:integer_stats_array;
      -- max queue count
    qtt:integer_stats_array;
      -- total time queue is used
    t:float_stats_array;
      -- average queue residence time
    qx:float_stats_array;
      -- average queue residence time
      -- based on qz
    qz:integer_stats_array;
      -- total zero entry count
      -- storage sna's
    se:boolean_stats_array;
      -- list to indicate if storage exists
    r:integer_stats_array;
      -- remaining capacity in storage
    s:integer_stats_array;
      -- current capacity of storage
    sa:float_stats_array;
      -- average storage capacity
    sc:integer_stats_array;
      -- total storage count
    sr:float_stats_array;
      -- utilization of storage
    sm:integer_stats_array;
      -- max storage
    st:float_stats_array;
      -- average holding time per unit
      -- transaction sna's
    te:boolean_stats_array;
      -- list to indicate if transaction exists
    pr:integer_stats_array;
      -- transaction priority level
    m1:integer_stats_array;
      -- transaction residence time in model
    ttc:integer;
      -- total transaction count
    ctc:integer;
      -- current transaction count
end record;
```

```
     sna:sna_record;
     procedure stats;
     procedure initialize_stats;
end sna_routines;

with transaction_routines;use transaction_routines;
generic
type t is limited private;
type p is access t;
type pa is array(index_type) of p;
package new_t is
     procedure new_task(a:in out p;exists:in out boolean);
     procedure initialize(a:in out pa);
end new_t;
```