Proceedings of the 1990 Winter Simulation Conference
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

CONCURRENT SIMULATION WITH ENTITY LIFE MODELING: AN AIRPORT SIMULATOR USING ADA

Pen-Nan Lee
Ravinder R. Purumandla
William Nehman

Department of Computer Science
University of Houston
Houston, Texas 77204-3475

ABSTRACT

Most simulation problems are inherently concurrent, and traditional approaches in trying to simulate concurrent problems by serializing them result in inefficient and complicated solutions. This paper presents the design and implementation of an Airport Simulator based on the entity-life model, the use of which results in a clear and intutive design. It is essential to choose a programming language with concurrent features for concurrent simulation. The Airport Simulator is implemented in Ada for its strong and effective tasking features.

1. INTRODUCTION

The development of modern parallel languages, such as Ada, Modula, Concurrent C, etc in recent years coupled with the advent of low cost microprocessors has made concurrent solutions to simulation problems more viable than ever. Such a concurrent solution in the Ada programming language could lead to significant improvements in the design, testing, run-time speed, code reusability, and life-cycle maintenance of the simulators.

The entity-life model [Sanden 1989] provides an efficient design principle that can be used to reduce a concurrent problem to a set of sequential problems. The entity-life model suggests that each independent thread of events in the real world be modeled as a task - the Ada language construct to represent an independent, asynchronous thread of events. In this article, the design and implementation of an airport simulator, based on the entity-life model is presented.

Most real world simulation problems can be naturally expressed as a set of physical entities, each having its own thread of events and interacting with the other entities in the problem environment when a need arises. In traditional approaches to simulation much effort is wasted in serializing a problem environment which is inherently concurrent. Thus when the entities in the problem environment have a strong time dimension, that is they often change state during the course of their existence, it is more logical to express them as software processes than as data. Prior to the development of languages such as Ada, incorporating software processes into a program meant that the programmer would have to write a language extension to support concurrency or even if the language did support a rudimentary form of concurrency he would have to use explicit commands to control their activation, interaction, etc. The problems with such approaches precluded the use of concurrent simulation techniques to most situations.

The Ada language on the other hand has concurrency intrinsic to the language and provides a logical, procedure-like interface between tasks, known as the rendezvous. Thus an Ada oriented design approach can simultaneously take advantage of tasking features as well as other modern language features available in Ada. These advances in language features combined with the new techniques available for design of concurrent software present a strong case for using concurrency in simulation, and also for using Ada as the simulation language.

While a few process oriented discrete event simulation languages, or packages [Thesen et al. 1983], are available to the programmer problems of documentation, time spent in learning a new language etc. are inhibitive. Moreover, the internal design for the simulation languages is not well documented in most cases. Such a situation combined with the fact that 90% of most simulation languages consist of general purpose code and 10% of application specific code written in the special purpose simulation language or a vendor supplied subroutine package [Thesen 1987], presents a case for writing simulators using general purpose languages as opposed to special purpose simulation languages. Portability considerations may also be taken into account while making a decision about the language to use. The Ada language [Gehani 1988], contains many unique language constructs and software engineering techniques such as data and process abstraction, object oriented design, information hiding, and has therefore been chosen for this simulation project.

2. SIMULATION MODELS

2.1 Classification

Simulation languages have been classified by the modeling viewpoint that can best be expressed using that language. These fall into the categories of Event, Transaction and Process-oriented. These are modeling viewpoints as opposed to language viewpoints of object-oriented modeling and data abstraction detailed in [Gomaa 1989]. In an event oriented view, the focus is on events, which alter the state of the system. The modeler can visualize the system as entities flowing through event sequences. Each type of event is represented in the simulation model by a software routine.In a transaction oriented view, the focus is on entity with attributes (transaction) flowing through the system. The simulation model describes all possible paths through the system, dealing with each entity independent of others. This view is natural to the modeler. The process oriented view focuses on the process, which is an event sequence. As such, the process includes events and the time delays between them. Each unique process is modelled by a process routine. Processes must become active and passive to model the flow and interactions of entities through the system. This view allows a natural correspondence with the abstract model for both the modeler and the programmer.

2.2 The Entity-Life Model

Various design strategies have been suggested for modeling a system as a set of concurrent processes, and most of these originate in the domain of Real time systems. The entity-life modeling approach suggested by [Sanden 1989], seems to be more applicable to problems outside the domain of real-time systems. Moreover, the other modeling theories tend to address function decomposition first, and then deal with task decomposition. The entity-life model on the other hand has task decomposition as its first objective, resulting in a clear and intuitive design which is isomorphic with respect to entities in the real world system that is being modelled.

Advantages of the entity-life model:

1. It results in a design which is isomorphic with the entities in the real world.

2. Each concurrent process becomes an information hiding unit, thereby encapsulating not only data structures representing the real world entities but also their current

state. Thus the model is inherently information hiding and we do not need to devote efforts to make the model information hiding.

3. If more than one instance of a particular kind of entity exists in the system it may be represented by a task type, with one instantiation of the task for each of the entities.

Melde and Gage [1989] suggest the following three phases for the entity-life model:

1. Identifying the entities in the real world model and mapping these onto tasks and other data structures.

2. The packaging phase deals with defining the modules and he interface between them. Also, further decomposition into packages takes place at this stage.

3. The implementation phase deals with the detailed elaboration of the modules defined in the previous stages.

The single most important feature of the Ada language is the tasking facility which allows the programmer to express concurrency and nondeterminism. These are the most important issues in modeling real world situations, and these have not been addressed previously. Hence, our approach in this simulation effort is to model the simulation in an entity-life model and to signify that the most important and rapidly changing objects in our simulation are active processes as opposed to passive data.

Such a model has the advantages of simplicity of design, increased run-time speed, portability and ease of development and testing effort over traditional methods of simulation which require serialization of concurrent activities, complicating the design and making the simulation slower.

According to the entity-life model, all physical entities, those that are existing in the real world and can manipulate their own state should be implemented as active elements of the language, i.e tasks. This is sometimes very important in a simulation because if the object can manipulate itself the simulation should be able to see its effects as soon as possible and any central control over it results in delaying that effect. For example if the active entity is an airplane, the airplane has to change its position continuously and display its position on a screen.

These tasks should manipulate themselves completely and should interact with other tasks only when absolutely necessary. This decreases global information in the system to a large extent. Each task should be able to do primitive tasks of input and output, on its own. This should be recognized, since if there are many concurrent tasks writing to the same screen at the same time the screen may become messy.

3. THE ADA LANGUAGE

3.1 Language Features

The following is a brief description of some of the important and unique language features of Ada used in this simulation.

(a) Tasks: Parallel processes in Ada are called tasks. A task executes independently and asynchronously, communicating with other tasks through message passing, using the ACCEPT and ENTRY statements. A task may call another task at it's ENTRY, and they can participate in bidirectional communication (termed rendezvous) after the called task accepts the entry call. A task is defined into two parts, task specification , and task body. The task specification presents the external interface to the task, and the body describes the actions performed by the task.

Features of the tasking Model

1. Provides a simple mechanism for communication without the need for explicit synchronization statements.

2. Provides for simultaneous bidirectional communication.

3. Instructions are executed by the called task on behalf of both the participating tasks.

4. The calling task must know the name of the called task.

5. A calling task can call only one other task and can only be in one entry queue of tasks waiting for a call to be accepted. A called task on the other hand can wait for any of many different types of calls and can have many queues of callers (one for each entry) waiting for its attention. The tasking model allows for the called task to have greater control over the rendezvous.

(b) Packages: Packages are information hiding units. They are defined in two parts, package specification and package body. Logically related entities and the operations that can be performed on them may be grouped into a package specification and the implementation details in the package body, thus presenting a high-level interface to users of the package while hiding the implementation details.

(c) Conditional rendezvous : The three kinds of select statements, Selective wait, conditional entry call, and timed entry call, facilitate conditional rendezvous between tasks in various situations.

(d) Access Types: Access types permit dynamic creation of objects of a predefined type, at runtime. When this delaration is used with a task type, it facilitates creation of any number of tasks of the predefined type at run-time.

(e) Termination: Ada provides for the terminate statement to ensure orderly and cooperative termination of tasks.

The working of the three kinds of select statements, the access type declaration and the terminate statement, are described in more detail in section 4.2.1, where they are used in the implementation of the airport simulator.

3.2 Ada and Other Languages

The most important strengths of Ada lie in concurrency but calling Ada a language suited only to real-time applications is far from the truth. It is a true general purpose language with all the traditional language features. It is portable like C, supports object-oriented programming principles like C ++ and has concurrent facilities like MODULA.

It is different from other languages because features to support concurrency are intrinsic to the language, i.e. it does not have any explicit task activation, suspend, resume or synchronization primitives such as message passing.

In addition to the language features mentioned in section 1, features such as strong typing , rigorous syntax checking, separate compilation, etc., save development costs. This will be the biggest criterion in future projects, as a result of the reverse trends of increasing software costs and decreasing hardware costs.

The Ada run time environment is responsible for run time scheduling of the tasks therefore, it can take advantage of as many processors as are available to the simulator, thus enabling porting of the simulator from a development system to an operating system which might have a different no. of processors.

4. THE AIRPORT SIMULATOR

4.1 Problem Description

The following is a short description of the airport which we wish to simulate.

(a) Airplane: There are two classes of airplane that visit the airport distinguished by their weight, light and heavy. Each airplane can be identified by a unique flight number and has a particular amount of fuel that keeps decreasing with respect to time while waiting to land, the rate of decrease being determined by its weight class.

(b) Airport: The airport has two runways, Heavy_Runway which can service planes of any weight class and Light_runway which can service light planes only. Each of the runways can be used for either takeoff or landing, but in case of emergency the landing airplane is given a higher priority. There are two runway controllers in the airport, one for controlling each of the runways. The job of the runway controller is to take the flight number of the requesting airplane from a queue and engage the airplane in a two-way conversation in order to provide the airplane with relevant weather, runway information, etc. and to guiae its landing or takeoff.

Each of the runway controllers has associated with it a queue of planes waiting to use the particular runway. The Air Traffic Controller(ATC) queues the requests into one of these two queues and the runway controllers dequeue each waiting plane and guide its use of the runway.

(c) Operation: Any airplane wishing to use one of the runways issues a land_request or takeoff_request to the ATC. The ATC then checks the queues associated with each of the runways and decides if the request can be serviced without a conflict with any of the planes that have already been accepted. If it can be accepted, it is put in one of the queues, else the airplane is informed that it has been rejected. The entity structure for the plane can be represented by an entity diagram as shown in figure 1.

4.2 Analysis, Design and Implementation.

The entity-life model vastly simplifies the design of the simulator, and makes the design easy to understand.

4.2.1 The Modeling Phase

The modeling phase begins with the identification of the major real world entities and mapping them into tasks and other data structures. A context diagram is a high level functional representation of the various entities involved in the system. Figure 2 shows the context diagram of the Airport simulator. The modeling phase identifies concurrency within the airport simulator.

Step 1: The following entities can be identified in the airplane simulator model.

1. Airplanes
2. Airport consisting of
   (a) Air Traffic Controller (ATC).
   (b) Runways (2) Heavy Runway and Light Runway,
   (c) Runway controllers (2),Light Runway Controller(LRC) and Heavy Runway Controller(HRC).
3. Terminal to receive instructions from and display results to.

These entities and their interactions can be represented by a further refined context diagram. Notice that all entities that have a time dimension should be modelled as tasks and are represented by shaded circles in figure 3.

Step 2: The next step is to further refine the design by identifying concurrency within each module. This process is done repeatedly until we cannot find further concurrency within each module.

These steps result in further splitting the airport task into the three tasks ,

1. Air Traffic Controller.
2. Light Runway Controller.
3. Heavy Runway Controller.

Step 3: At this stage we have to make decisions on abstract entities which have a time dimension. This step results in one more task, the Random request Generator to randomly generate flights for the simulator to process. Statistics task collects statistics on all planes and analyses them. Dynamic Queues may also be represented by tasks since they have a definite time dimension.

1. Random Request Generator -- Randomly generates flights.
2. Statistics

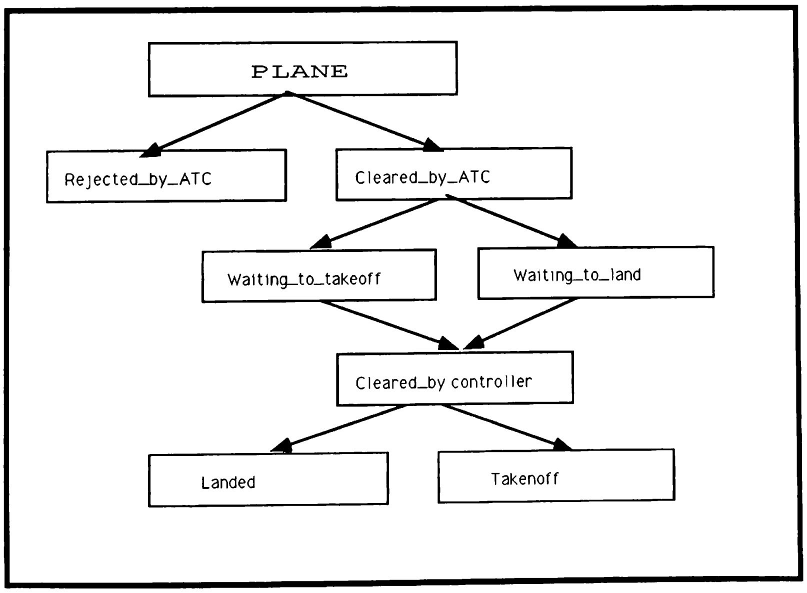


Figure 1. Entity Structure for PLane

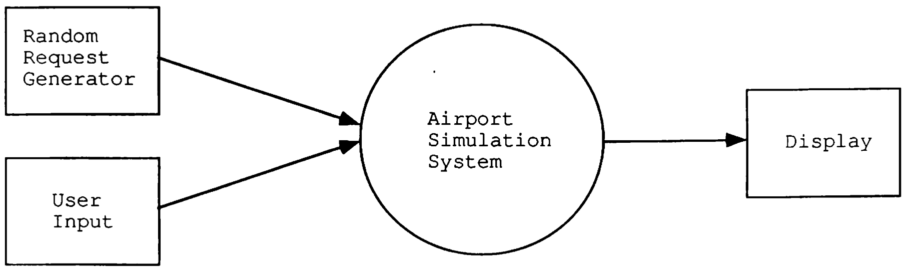


Figure 2. Context diagram for an Airport simulator

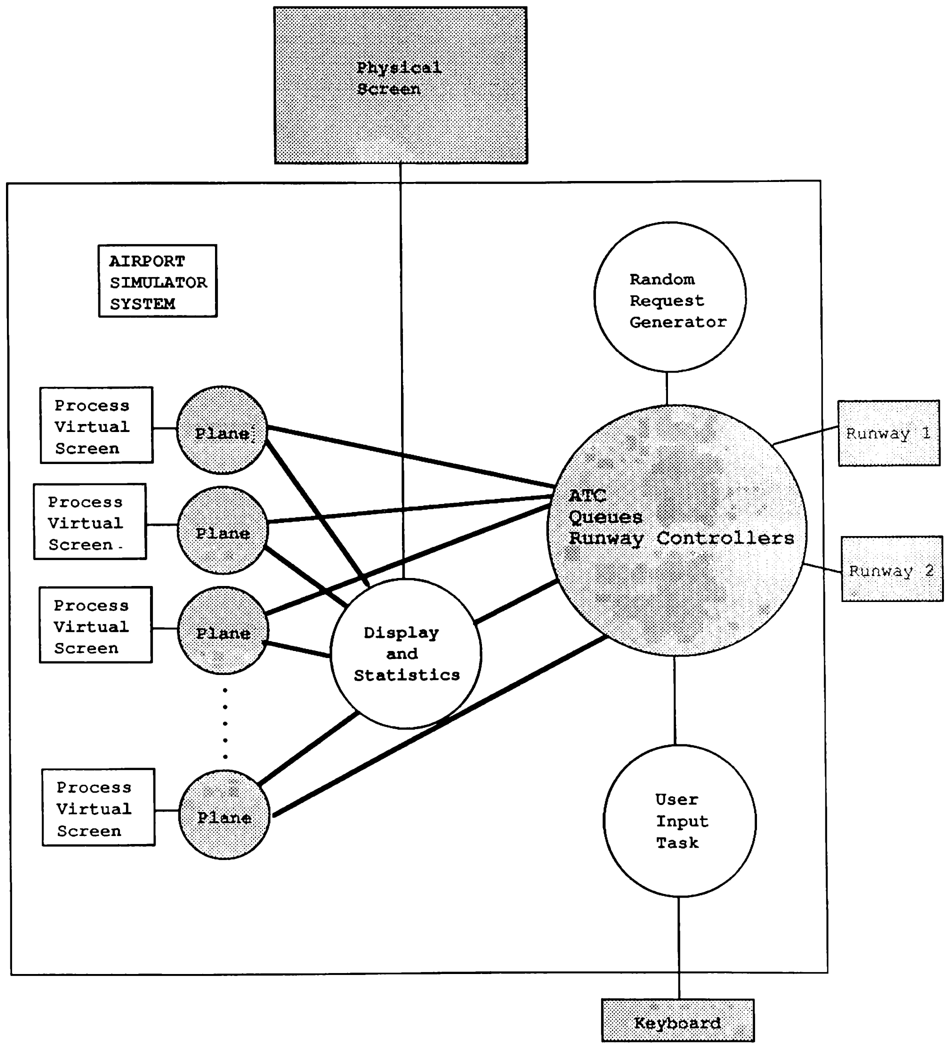


Figure 3. Context diagram for Airport Simulator.
Shaded Objects represent real world entities.
Unshaded objects represent abstract entities.

- Collects statistics on all planes.

3. H_Queue
   - Queue for the heavy controller.
4. L_Queue
   - Queue for the light controller.
5. H_Stat
   - Buffer task to record status of H_queue
6. L_Stat
   - Buffer task to record status of L_queue

Step 4: The next step is to identify tasks to handle specific I/O devices. This results in two tasks

1. User Input task.
2. Display Task.

At this stage, some of the tasks may be combined to simplify the design.
We combine the display task with the statistics task, to yield a simpler design.

4.2.2 The Packaging Phase

The packaging phase consists of defining the way tasks, packages, subroutines and functions are grouped together so that the resulting model is logical and follows the objectives of information hiding and data abstraction.
This phase follows three steps.

1. Determine interfaces for communicating units, such as tasks and packages.
   (a). Define entry points for each task.
   (b). Determine parameters and in/out modes for each rendezvous.
   (c). Specify exceptions raised and Handled.
2. Package global data types and constants.
3. Package hardware dependent features for each device.

This phase results in four packages for the Airport simulator,

Air_Facilities, Ground_Facilities, Random_Requestor and Support_Package.

Package Air_Facilities
- specification and implementation of the entity Airplane.

Package Ground_Facilities
- specification and implementation of the remaining entities, of the simulator.

Package Random_Requestor
- the task Generator which on request from the user randomly generates flights to be processed by the simulator.

Package Support_Facilities
-- packages that help in building Air_Facilities and Ground-Facilities, such as screen management features, Math functions, Random no. generation functions etc.

Specification of package GROUND_FACILITIES.

```
package GROUND_FACILITIES is
        task ATC is
                entry start_atc(P:in INTEGER);
                entry LAND_REQUEST(A:in out
                        PLANE_REC;ADDR:in T_ID);
                entry TKOFF_REQUEST(A:in out
                        PLANE_REC;ADDR:in T_ID);
        end ATC;
        task H_QUEUE is
                entry START_H_QUEUE(P,D1,D2,D3:in
                        INTEGER);
                entry ADD_TO_Q(A:in out PLANE_REC;ADDR:in
                        T_ID;Q:in INTEGER);
                entry NEXT_JOB(ADDR:out T_ID;H1:out
                        fly_time;h2:out
                        FLY_TIME;R_STATUS:out STATUS);
        end H_QUEUE;
        task H_STAT is
                entry CLEARED(H1:in FLY_TIME;H2:in
                        FLY_TIME);
                entry READ_STATUS(w1:out fly_time;w2:out
                        fly_time;T:out FLY_TIME);
        end H_STAT;
        task HRC is
                entry START_HRC(P:in INTEGER);
                entry wakeup;
                entry FINISHED;
        end HRC;
        task L_QUEUE is
                entry START_L_QUEUE(P1,D1,D2,D3:in
                        INTEGER);
                entry ADD_TO_Q(A:in out PLANE_REC;ADDR:in
                        T_ID;Q:in INTEGER);
                entry NEXT_JOB(ADDR:out T_ID;L1:out fly_time;
                                                L2:out
FLY_TIME;R_STATUS:out STATUS);
        end L_QUEUE;
        task L_STAT is
                entry CLEARED(L1:in FLY_TIME;L2:in
                        FLY_TIME);
                entry READ_STATUS(w1:out fly_time;w2:out
                        fly_time;T:out FLY_TIME);
         end L_STAT;
        task LRC is
                entry START_LRC(P:in INTEGER);
                entry wakeup;
                entry FINISHED;
        end LRC;
        task DISPLAY is
                entry START_DISP(d:IN int_array);
                entry CLEARED_BY_CONTROLLER(A:in
                        PLANE_REC;N:in INTEGER);
                entry CLEARED_BY_ATC(A:in PLANE_REC;S:in
                        INTEGER;N:out INTEGER);
        end DISPLAY;
        task USER_INPUT is
                entry START_INPUT(P:in integer);
        end USER_INPUT;
end GROUND_FACILITIES;
```

Specification of package AIR_FACILITIES:

```
with PLN_TYPS; use PLN_TYPS;
FIFO;use FIFO;
with QUEUES; use QUEUES
with SMG; use SMG;
```

Together , these packages define the objects airplane and Priority Queues, and with operations that can be performed on them such as adding a plane to a queue, deleting from a queue.The SMG package defines screen management functions.

```
package AIR_FACILITIES is
        type AIRPLANE;
        type T_ID is access AIRPLANE;
        task type AIRPLANE is
                entry INITIALIZE(A:in PLANE_REC; ADDR:in
                        T_ID);
                entry REQUEST_GRANTED(S:out INTEGER);
                entry REQ_REJECTED;
        end AIRPLANE;
        NEW_AIRPLANE,ANOTHER_AIRPLANE:T_ID;
end AIR_FACILITIES;
```

4.2.3 The Implementation Phase

This phase involves expansion of task, package, subroutine and function bodies. The following three pages contain a concise description of the package body GROUND_FACILITIES.

task ATC: This task is responsible for initial communication with Airplanes, and to decide whether they are accepted or rejected and if accepted, the request is put in the appropriate queue. It is called through the entry points Land_Request and Tkoff_Request.

```
Loop
    Select
        accept  LAND_REQUEST(B: in out  PLANE_REC;
                     ADDR: in TASK_ADDR );
          - Check status of Queues and put the flight in one of
          - the queues if the request can be serviced, else notify
          - flight.If it is a heavy plane, only H_Queue needs to
          - be checked.
      or

          accept  TKOFF_REQUEST(B: in out  PLANE_REC;
                              ADDR: in TASK_ADDR );
           - Add plane to appropriate queue if possible.
      or

          terminate;
      end select;
End loop;
End ATC;
```

task H_QUEUE: This is the dynamic queue for the Heavy Runway Controller(HRC). It acts as a buffer between the HRC and ATC. ATC adds to the queue and HRC removes flights for processing. It is accessed by ATC through the entry point ADD_TO_Q, and by the HRC through the entry point NEXT_JOB. Implementing this dynamic queue as a task serves three purposes:

1. Ensures that the queue is accessible at all times.
2. Ensures mutual exclusion among the tasks accessing the data.
3. The guards used in the select statements also ensure priority access for the runway controller, as it is the prime resource in the system, and is to be kept busy all times.

```
loop

    select
        accept  NEXT_JOB( ADDR:out T_ID;H1:out
              FLY_TIME;H2:outFLY_TIME;R_STATUS:out
                                        STATUS) do
              if  (ANY_PLANE(HRC_Q) = TRUE)  then
               - IF H_QUEUE is not empty , return next flight to be
               - serviced by   the controller
              end if;
        end  NEXT_JOB;
      - Update number of planes accepted ,landed and takenoff.

    or

        when  H_queue is not full   =>
        accept ADD_TO_Q(A:in out PLANE_REC; ADDR:in
                    T_ID;Q :in INTEGER) do

      - accept another plane from ATC and add to Queue.
      - if Queue status has changed from empty wakeup HRC.

        ADD(HRC_Q,P1,B);

      - Add to the queue, with priority specified by ATC.
      end ADD_TO_Q;

      - Update number of planes accepted for take off and no. of
      -  planes accepted for landin Update the virtual screen of the
      - process.

      H_screen_update;
    or
        terminate;
  end select;
End_loop;

end H_QUEUE;
```

tasks HRC and **LRC:** These are the two runway controllers, that control the two runways. These two tasks control the runways which are the most important resources of the airport and hence should never be kept idle. The controller calls the airplane to signal it to start landing, and also guides it during the takeoff - landing process.

```
loop
  Select
          when AWAKE = NO =>
                  accept wakeup do
                    - wakeup
                   end wakeup;
        - While queue is not empty get next flight no. in queue
        - call the plane. Wait until it responds by issuing an entry
        - finish entry.
        - If there are no more planes in the queue, i.e
        - when  AWAKE= NO,
        - wait until a wakeup call is given by H_QUEUE.
        - This prevents HRC from accessing an empty queue.

        loop
          - Get next job from H_Queue.
            h_queue.next_job(baddr,hwait1,hwait2,awake);
            If  AWAKE = NO  then
                - Que is empty, wait for wakeup call.
                exit;
              else
                - Call the flight and get its screen ID.
                        BADDR.REQUEST_GRANTED(S);
                - Paste its screen on the top of the screen.
                MOVE_SCREEN(S,P_1,4,21);
                - Inform H_STAT of the status of H_QUEUE.
                H_STAT.CLEARED(HWAIT1,HWAIT2);

                - Accept an acknowledgement from plane and
                - get flight no. from H_Queue.
               accept FINISHED;
             end if;
         end loop;
    or
       terminate;
     end select;
   end loop;
end HRC;
```

tasks H_STAT and **L_STAT**: These tasks are introduced to remove the dependency among tasks that would otherwise result in starvation or perhaps deadlock.

The ATC has to get information from the controller on the status of the queues and runways, and the time it will take for the controller to service the present takeoff - landing that it is handling. In order to obtain this information the ATC would have to call the Queues and the controllers to find out about their positions, each time it receives a request from a plane. This could lead to a situation in which the ATC is always waiting for one of the controllers and hence cannot accept requests until it has received a response from both controllers and both queues. The H_STAT task is always available to the ATC, while the HRC is servicing the plane and therefore does not delay the ATC.

```
loop
   Select
        accept CLEARED(H1:in FLY_TIME;H2: in FLY_TIME)  do

        - Get information on status of H_QUEUE and the
        - time at which the last plane to be serviced by controller
        - started to use the runway.

        end  CLEARED;

   or
```

```
        when CLEARED'COUNT = 0 =>
         - When Controller is not waiting for rendezvous with
         - this task.

         accept READ_STATUS(W1:out FLY_TIME;W2:out
                        FLY_TIME;T:out FLY_TIME) do

          - Calculate wait time for the queue based on information
          - provided by controller and time elapsed since then
          - and return it to ATC.

        end READ_STATUS;
    or
        terminate;
    end select;
   end loop;
 end H_STAT;
```

task USER_INPUT: This task is a low priority task and facilitates user input. Since it is a simulation, the user should be able to give his instructions to the system at all times, without stopping the system, e.g. he should be able to create more requests, if he wishes to, while the simulation is running.

```
loop
   - Always looks for input from the keyboard.
   - Gives the option to run simulations by time, or by no.
   - of simulations,
   - and calls the generator to generate planes.
end loop;
```

Package RANDOM_REQUESTOR
This package contains one task - Generator, which accepts input from User_Input task.

task GENERATOR: This is a background task whose sole purpose is to randomly generate requests for the simulator, on demand by the user. It is important that this may not communicate with the other tasks in any way than by producing requests, in order that the nondeterminism and randomness of the simulation are preserved.

```
loop
  Select
        Accept GENERATE(NO_OF_SIMS:in
                        INTEGER;inp_type:in INTEGER) do
              - Each time it is called it loops for a random number
              -  oftimes to generate flight requests, and makes one
              - task for each flight generated and initialises with
              - random values of fuel, flight_no etc.
        end GENERATE;
    or
        accept DONE do
              - No more planes to generate.
              exit;
        end DONE;
   end SELECT;
 end loop;
end GENERATOR;

package body  AIR_FACILITIES  is
  task body  AIRPLANE is
        accept INITIALIZE(A:in PLANE_REC; ADDR:in T_ID) do
              - Initializes the plane with values of fuel, request type
              - weight class etc, generated by  the Random no.
              - generator.
        end INITIALIZE;
        - Creates a virtual screen for itself.
        - Request ATC for landing or takeoff.
        - Informs display task of its status.
        Select
              accept  REQUEST_GRANTED(S:out INTEGER) DO
                    - If called by controller to use runway, inform
                    - display task.
              end  REQUEST_GRANTED;

                 - delay for time required for plane to land.
                 - Delete virtual Screen
                 - DO DEALLOCATION
                 - Reclaims the storage allocated for
                 - the task.

        or
              accept  REQ_REJECTED do
                    - DO DEALLOCATION;
              end REQ_REJECTED;

                 - Wipe virtual screen
                 - DO DEALLOCATION
        end select;
begin
        null;
 end AIRPLANE;

end  AIR_FACILITIES;
```

The hierarchy of packages results in an order of compilation as shown in figure 4. The package PLN_TYPS implements the airplane entity.

The Package FIFO uses the airplane entity described in PLN_TYPS to implement a First-in First-out queue. Package QUEUES uses packaage FIFO to provide a priority queue, and so on. An arrow from the body of FIFO to the package QUEUES signifies that the body of FIFO should be compiled before the package QUEUES can be compiled.

Figure 5 gives a detailed description of the packages used and their relationship. A rectangle represents a package, and encloses the functions exported by the package.

5. PROBLEMS ENCOUNTERED

A major problem with the entity model is that each of the concurrent processes requires much virtual memory. When the number of real world entities is inhibitive, some real world entities which do not change state frequently have to be modelled as data instead of tasks. Moreover, as the number of active tasks increases the speed of the system decreases.

With the introduction of concurrency the programmer has to deal with issues peculiar to concurrency, such as circular wait. Circular wait is a situation in which two or more tasks are waiting for resources held by each other, and none of them will proceed further until the resources are available to it. A simple analogy would be one in which three persons, A, B and C are trying to call each other. A is trying to call B, B is trying to call C and C is trying to call A. In the process none of them is able to communicate with the other.

6. CONCLUSIONS

The concurrency features of Ada, combined with the design principles of the entity-life model proved to be very appropriate to the Airport simulation problem, which is inherently concurrent. The tasks resulting from the design based on the entity-life model were mostly isomorphic to the real world entities they represent and thus helped make the design simpler in addition to capturing real world concurrency which is so vital to many applications. The modern language features of Ada, particularly its concurrency mechanisms, and design principles such as entity-life modeling clearly demonstrate the viability of Ada as a language for simulation.

REFERENCES

Gehani, N. (1988), *Ada - An Advanced Introduction,* Prentice Hall Inc., Englewood Cliffs, N.J.

Gomaa, H. (1989), ``Structuring criteria for real time design,''*IEEE 11th International conference on software engineering*, 152-164.

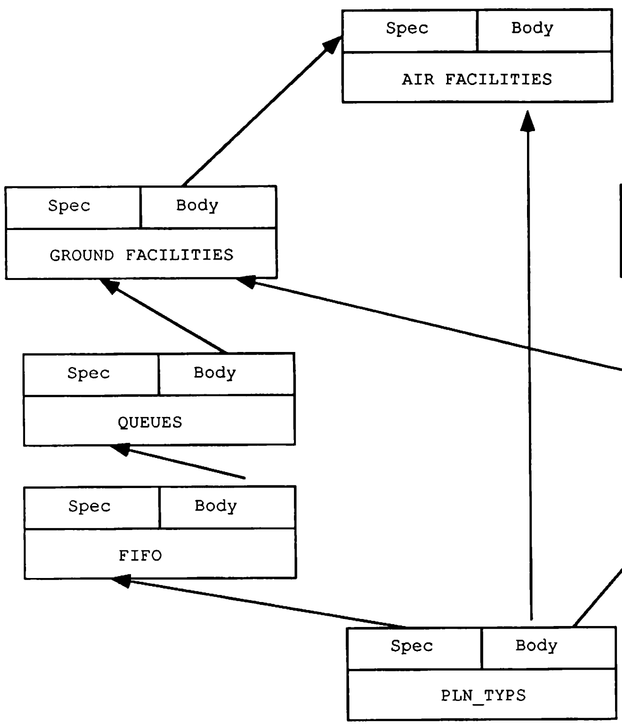


Figure 4. Airport Simulator System Dependency Graph

Melde, J. E. and P.G.Gage (1989), ``Ada Simulation Technology - Methods and Metrics,"*Proceedings of t he seventh Annual Simulation Symposium.* 11-26.

Nielsen K.W, and K.Shumate (1987),``Designing Large Real-Time systems with Ada,"*Communications of the ACM* , 30, 8, 695-715.

Powers, W.S. and T.Nute (1985),``Implementing a simulator as a set of Ada tasks,"*Proceedings of t he Eastern Simulation Conference,* March 85, Norfolk Virginia, 7-12.

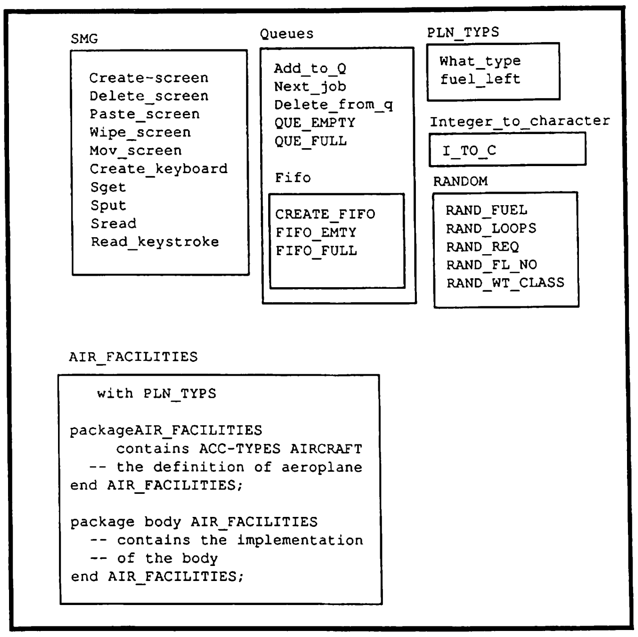
Sanden,B. (1989),``An Entity-life Modeling Approach to Design of Concurrent Software,"*Communications of the ACM* 32, 3, 330-343.

Sanderson, D.P. and L.L.Rose (1989),``Object oriented Modeling using C++," *Proceedings of the Annual Simulation Symposium,* 143 - 149.

Thesen A., Grant H., and D.W.Kelton (1983),``A process-Oriented SimulationPackage based on Modula-2,"*Proceedings of the 1987 Winter Simulation Conference* , Dec 14-16 1987, Atlanta, Georgia, 165-169.

Thesen A. (1987),``Writing simulations from scratch,"*Proceedings of the 1987 Winter Simulation Conference* ,152-156.

United States Department of Defense (1983),``Reference Manual for the Ada Programming language" ANSI / MIL - STD-1815A-1983, United States Department of Defense, Washington, D.C.



```
  with PLN_TYPS

packageAIR_FACILITIES
    contains ACC-TYPES AIRCRAFT
  -- the definition of aeroplane
end AIR_FACILITIES;

package body AIR_FACILITIES
  -- contains the implementation
  -- of the body
end AIR_FACILITIES;
```

Figure 5. Packages