Proceedings of the 1983
Winter Simulation Conference
S. Roberts, J. Banks, B. Schmeiser (eds.)

INTERACTIVE MODELING AND SIMULATION OF TRANSACTION FLOW OR NETWORK MODELS USING THE ADA SIMULATION SUPPORT ENVIRONMENT

Heimo H. Adelsberger
Abteilung fuer angewandte Statistik
und Datenverarbeitung
Wirtschaftsuniversitaet Wien
Augasse 2-6
A-1090 Vienna, Austria

The Ada Simulation Support Environment (ASSE) is a software system, with the purpose to support the development and maintenance of simulation models written in Ada throughout their life cycle. We describe here the transaction flow or network part of the ASSE, which allows to build models like in GPSS or SLAM. Our view of such models is slightly different from that of the above mentioned languages, which is demonstrated in detail by the server/resource process. The design stres ses modular top-down development using submodels. Models can be developed and tested interactively.

1. THE ADA SIMULATION SUPPORT ENVIRONMENT

The main guide lines for the design of the ASSE have been:

- package concept: instead of introducing a new simulation language, which fulfills the requirements of modern software technology as well as the state of the art of simulation, we gave preference to a system of packages.

These packages are well tuned together and can be used at two levels:

- on a low level, where a deeper insight of all concepts is necessary and where support is provided for basic tasks like queue management, random variates, entities-attributes-sets management and simulation control (combined modeling: continous, discrete event, activity scanning and process interaction including transaction flow or network modeling).

- on a high level, where the usage is very user-friendly and decidely simple, and where the stress lays on the fact, that simulation is a human activity, affected by a wide range of different techniques, where the "man in the middle" has to be supported and has to be freed from all technical details not concerned directely with the application domain.

The packages on the high level can be divided in two classes:

the actual simulation subsystems like

model_design
model_verification
model_documentation

and the support systems like

data_base_management
statistical_analysis
graphics
screen_printer_IO

For the latter packages it is not intended to provide these packages themselves, but to provide interfaces to standard systems, which - as we hope - will soon be available in Ada (in package form!).

We have presented the overall design of the ASSE and hereby the discrete event approach more in detail in "ASSE - Ada Simulation Support Environment" (Adelsberger 1982). We have dealt with transaction flow or network modeling in our paper "A Structured and Modular Approach to Transaction Flow Models" (Adelsberger 1983a). We catch up this subject here again focusing our discussion to the following topics:

- submodels
- the server process
- interactive design and testing of transaction flow (network) models

We demonstrate our ideas by an example, which is an extension of an example given by T. Schriber (1974): "Inspection Station on a Production Line". A Q-GERT version of the example can be found in Pritsker (1977), a SLAM version in Pritsker (1979). A reader interested in technical details of the implementation is refered to the paper "Transaction Flow Models in Ada: Technical Background" (Adelsberger 1984).

2. TRANSACTION FLOW OR NETWORK MODELING

The languages GPSS and SLAM (the network part) are presently dominating the area of transaction flow or network modeling. Both languages provide the typical constructs for this type of systems. The notation offered by these languages carries directly the corresponding natural conceptual framework. The rather high semantic level (in respect to the application field) has the advantage to speed up model development; a certain restriction of flexibility has to be put up with.

We consider both languages to be rather poor as programming languages (compared e.g. with Pascal or Ada). The syntax of GPSS is designed like an assembler. It has many deficiencies such as bad control structure, low redundancy (using numbers where names would be desirable). The improvements of the last years on GPSS (GPSS/H) have diminished these shortcomings, but could'nt change the basic conceptual weakness. SLAM suffers from the fact, that it is based on FORTRAN, so all weak points recognized in FORTRAN, are valid for SLAM.

But even in respect to their application areas we find that these languages have their conceptual deficiencies. As examples we mention the following aspects:

(a) GPSS and SLAM do not support submodels. Experience in the field of general programming has shown, that modularity and top-down design has improved program development. As examples, complexity is reduced and readability is improved. The same can be expected, if submodels can be defined. We are going to present a way to implement submodels.

(b) We think, that both languages would profit from a more precise approach. We think, that in such a framework it has to be clearly distinguished between what can be represented in a network in form of nodes (blocks) and what exists in the model, but has no direct counterpart in form of a node or block. Both languages tend to implement the latter in form of nodes (blocks), which leads to unnatural representations of the given real system. SLAM has withdrawn from the GPSS principle to represent everything in network notation. SLAM makes more use of control statements outside the network: so one can set the time to start and to end the simulation in a natural form via an "INITIALIZE" statement. A GPSS user has to define for the same purpose a separate network, producing a clock transaction to stop the simulation run. Furthermore SLAM provides information about the queues, which are used in the network, via control statements like PRIORITY. Finally, SLAM separates the general information about resources and gates from the actions, taken at node arrivals. This general information is given in form of "blocks" (not to be confused with the GPSS blocks!), to which references are made in nodes like "AWAIT", "OPEN" or "CLOSE".

In our approach we are working out clearly the difference between this overall information and the action which has to be taken, when an entity arrives at a node. We demonstrate this in detail in our description of the service process. As we will see, our conception of the service process enables the user to implement simple situations easier than in SLAM or GPSS, but it enables also to implement situations in a straigtforward form, which can only be implemented in GPSS or SLAM in a tricky or complicated form.

A further advantage of the Ada Simulation Support System is the fact that a modeler can dispose over the complete expressive power of Ada.

3. THE ASSE NETWORK PART

3.1 Models (Submodels)

A model (submodel) is described by means of its static and dynamic structure. The static structure is given by global variables, entity attributes and the different processes like servers, resources, queues etc. The dynamic structure is given in form of a network.

3.2 The Processes (Static Structure)

The processes are listed here; some of them are described in more detail beneath:

QUEUE	SERVER/RESOURCE	MATCH
CREATION	GATE	
TERMINATION	ACCUMULATE	

3.3 The Network (Dynamic Structure)

A network is decribed like a task. The network specification contains the enter_ and outlet_declarations (in analogy to the entry_declaration of a task). They denote places where entities can enter or leave a network. This comprises the generation (creation) and termination of entities and the transfer of entities from one network to another one.

The required actions concerning these processes are effected in the statement part of the network body via entry calls or accept statements like:

REQUEST	WAIT	PREEMPT
SERVICE	RELEASE	CANCEL
OPEN	CLOSE	

There are predefined processes in the simulation_standard_package, which have not to be declared seperately and which can be used to simulate the passing of time or the splitting of entities:

HOLD SPLIT

There are no specific requirements for ASSIGN nodes or blocks. Most valid Ada statements may be used in the network body. Standard numerical attributes (SNA's) are defined for processes and entities (transactions) like in GPSS . These SNA's can be used wherever variables can be used.

There are no specific control statements (like the TRANSFER block in GPSS or the ACTIVITY branching in SLAM), because the usual control constructs of Ada may be used: LOOP, IF-THEN-ELSE, CASE.

3.4 Modularity

The ASSE network part supports in excellent form top-down design and modularity: It is possible to

define submodels and generic submodels to construct a model out of different submodels.

Submodels are easy to understand: they act like subprograms (procedures) in a program. Generic submodels can be seen as templates of model units, and can be parametrized. Instances (meaning copies) of the generic submodels can be made, and can then be used in a model.

A model is organized like a generic Ada package. It has to be distinguished between the following:

- what can be seen and can be used from the outside
- what is hidden and only known inside the submodel
- what must be provided from the outside

This corresponds in an Ada package to:

- package specification
- package body
- generic parameters

With models we speak about:

- model specification
- model body
- model parameters

Normally, the model specification consists of the enter declarations, but could contain server and queue declarations. The model body consists of the specific server and queue declarations for the submodel and the network. This can be considered as a black box from the outside. The model parameters are normally the outlet points of the network. An outer model has to take care of the entities leaving a submodel at an outlet point. This can be done via a termination process or by a link (transfer) to another enter point visible in the enclosing model. A submodel can be compiled separately, which speeds up the whole development process.

3.5 Global Variables

Global variables are attributes of the whole system. They are declared as Ada objects in the declarative part of the model.

3.6 Entities

Entities are declared in three stages. First the different entity kinds are introduced in enumeration form. Then the attributes of the entity are declared as a record construct with the entity_name as the discriminant and a variant part. Finally 'entity' is introduced as an access type for the attributes. The actual entity is called current_entity.

Example:

```
type entity_name is ( haircut_only_customer,
   shave_and_haircut_customer);

type attributes (kind : entity_name ) is
   record
      info: hidden_info;
       age: integer;
      service_time_haircut: float;
      case kind is
         when shave_and_haircut_customer =>
            service_time_shave: float;
         when others =>
            null;
      end case;
   end record;

   type entity is access attributes;
   current_entity: entity;
```

A component 'info' of the limited private type 'hidden_info', common to all variants, has to be declared. This type is provided in the entity - attributes - set manager and serves to control the entities (create, insert etc.). (For a more detailed discussion see Adelsberger 1982.)

A basic idea is to provide each entity with its own processor executing the node arrivals. The task performing in this way is also a component of the field 'info'.

3.7 Simulation Control

The outermost model body has to control the simulation runs. This is done via procedure calls. The model section starts with the word "SIMULATE". The basic ideas are like in SLAM. The more important procedures are:

```
procedure general_information
   (project_name : string; project_date : time);

procedure seeds (stream_number: integer;
                 initial_value: integer;
                 reinitialization: boolean);

procedure initialize
   (starting_time: float := 0.0;
    ending_time  : float;
    initialize_variables,
    initialize_queueing_system,
    clear_statistical_arrays: boolean := true);

type monitor_option is (summary, queues, clear);

procedure monitor
   (option : monitor_option;
    first_execution,
    time_between_execution : float);

procedure trace
   (option : trace_option;
    first_execution,
    last_execution : float);

procedure start_simulation;
```

4. THE PROCESS DESCRIPTIONS

4.1 Queue

Queues are described in more detail in a recent paper by the author (Adelsberger 1982). The parameter describing the queue are: the queue capacity, the initial number of entities in the queue and the priority rule for the queue. A full queue can block server/resource processes. In this case a list of their names has to be provided.

Additional declarations:

```
type queue_priority is (FIFO, LIFO, HVFO, LIFO)
```

Initialization parameters:

```
queue_capacity:    natural := integer'last;
initial_occupancy: natural := 0;
priority : queue_priority  := FIFO;
block: array (natural range <>) of
          server_resource_name;
```

SNA's:

```
queue_size:          natural;
queue_full:          boolean;
entity_count:        natural;
minimum_content,
maximum_content:     natural;
average_residence_time: float;
```

4.2 Creation Process

Entities can be created and inserted in a network. A procedure can be supplied to initialize the attributes of the entity. The maximum number of creations, the time for the first creation and the time between subsequent creations can be stated. The place, where the entity is to be inserted (the ENTER node) has to be fixed.

Initialization parameters:

```
entity_kind              : entity_name;
initialization           : procedure_name;
time_of_first_creation   : float :=0.0;
time_between_creations   : float;
maximum_number_of_creations :
   integer :=integer'last;
enter_name               : enter_node_name;
```

Usage:

```
accept enter_name;
```

SNA's:

```
count : integer;
```

Example:

Declaration:

```
procedure customer_initialization is
begin
   customer.year_of_birth := current_year;
end customer_initialization;

customer_generation : creation
    ( entity_kind              => customer,
      procedure_name           =>
         customer_initialization,
      time_of_first_creation   => 0.0,
      time_between_creations   =>
         uniform (10.0,20.0,1),
      max_number_of_creations  => 100,
      enter_name               => entrance );
```

Application:

```
accept entrance;
```

4.3 Termination

Entities can be removed from the network. The termination process counts the removed entities. A number can be specified and as soon as that number is reached, the simulation is stopped.

Initialization parameters:

```
stop_simulation: integer:= integer'last;
```

SNA's:

```
count: integer;
```

Usage:

```
outlet(process_name);
```

Example:

Declaration:

```
exit_door: termination;
emergency_exit: termination (10);
```

Application:

```
outlet(exit_door);
outlet(emergency_exit);
```

4.4 Server or Resource Process

This process has two forms of interpretation: It can be regarded as a server process with one or more parallel or identical servers or it can be interpreted as a process to get access to a resource. The requests of servers/resources are queued, different queues can be involved. The list of queues where the requests are scheduled has to be provided. An optional select criterion states, from which queue the next entity is chosen for the service, if entities wait in more than one queue. The default criterion is cyclic. Servers (resources) can be preempted.

Additional declarations:

```
type server_status is (idle, busy);
type select_alternative is (
   given_order, cyclic,
   random, assembly_mode_option,
   largest_average_number,
   smallest_average_number,
   longest_waiting_time,
   shortest_waiting_time,
   largest_queue, smallest_queue,
   largest_remaining_capacity,
   smallest_remaining_capacity);
```

Initialization parameters:

```
capacity: natural:= integer'last;
   -- or number of parallel servers
preemptable: boolean:= false;
queue_name_list:
   array (positive range <>) of queue_name;
select_criterion: select_alternative:=cyclic;
```

SNA's:

The SNA's can be called in two forms: indexed, denoting a specific server (resource unit), or

without index. In that case the attributes for the whole process are given. The range of the index is 1 .. capacity.

```
status: server_status;
count,
preemption_count : natural;
preempted: boolean;
remaining_processing_time,
consumed_processing_time: float;
current_capacity,
remaining_capacity: integer;
utilization,
average_holding_time: float;
```

Usage:

Interpretation as a server: An entity requests service from a server at a REQUEST node in the network. A queue is determined to keep track of the incoming requests. A priority value can be submitted, which can be used in connection with the PREEMPT node. If a server is idle, a connection between this entity and the server is established. If all servers are busy, this request is queued, but in contrary to GPSS and SLAM (!!!), it doesn't prevent the entity from moving on. Only reaching a WAIT_FOR_SERVER node prevents the movement of the entity, if it could not get service in the meantime. The server is released at a RELEASE node.

Interpretation as resource process: The indicated number of units are requested at the REQUEST node. The entity waits when reaching a WAIT_FOR_RESOURCE node and still not enough units of the resource are available. The indicated number of units is returned at the RELEASE node.

```
REQUEST (name: process;
   at_queue: queue_name;
   units_requested: integer :=1;
   priority: float := 0.0);

WAIT_FOR_SERVER (name: process);
WAIT_FOR_RESOURCE (name: process);

RELEASE (name: process;
         units_released: integer:= 1);
```

A WAIT node combines the function of a RELEASE and a WAIT_FOR_SERVER (WAIT_FOR_RESOURCE) node. This WAIT node corresponds in case of a server exactly to the SLAM 'QUEUE' node and to the GPSS 'SEIZE' block, if the 'SEIZE' block is surrounded by a 'QUEUE' and 'DEPART' block.

```
WAIT (name: process;
   at_queue: queue_name;
   units_requested: integer :=1;
   priority: float := 0.0);
```

It is possible to provide the server with the information about the length of the service at the REQUEST node. Two cases are distinguished: The end of the service is indicated via a service time or via a condition, which has to be true to end the service. The WAIT_END_SERVICE node is important in this context, which prevents an entity from moving, if the requested service is not finished. The RELEASE node is superfluous in this case and its usage would be erroneous.

```
REQUEST (name: process;
   at_queue: queue_name;
   with function_name return event_time;
   units_requested: integer :=1;
   priority: float := 0.0);

REQUEST (name: process;
   at_queue: queue_name;
   with function_name return boolean;
   units_requested: integer :=1;
   priority: float := 0.0);

WAIT_END_SERVICE (name: process);
```

The combination of a REQUEST node of this second kind with a WAIT_END_SERVICE node is the SERVICE node. This single SERVICE node replaces the complete GPSS sequence "QUEUE - SEIZE - DEPART - ADVANCE - RELEASE" or the SLAM construct "QUEUE + Service ACTIVITY".

```
SERVICE (name: process;
   at_queue: queue_name;
   with function_name return event_time;
   units_requested: integer :=1;
   priority: float := 0.0);

SERVICE (name: process;
   at_queue: queue_name;
   with function_name return boolean;
   units_requested: integer :=1;
   priority: float := 0.0);
```

The capacity of the resource (the number of parallel servers) can be changed at an ALTER node.

```
ALTER (name: process;
       units);
```

A service (resource) request may be cancelled. If the service has already started, the service is interrupted immediately. The consumed service time is saved and can be used by the entity. For a resoure process, the units are returned.

```
CANCEL (name: process);
```

It is possible to preempt a server from service, if the priority of the incoming entity is higher than that of one of the served entities. The server with the entity having the lowest priority is preempted. The action takes place at a PREEMPT node. If preemption is impossible, the incoming entity waits in the specified queue. If preemption is possible in a model, the preemption flag of the server has to be checked. If the preemption flag is not checked and an entity tries to move on, a programming error is raised. The remaining service time of an entity is saved and can be used by the preempted entity via the numerical attribute 'remaining_processing_time'.

```
PREEMPT (name: process;
   q: queue_name;
   units_requested: integer := 1;
   priority: float);
```

Examples:

(1) 3 identical servers, 1 place in the model to wait for service, therefore no select criterion.

Declaration:

```
clerk : service ( capacity => 3,
          queue_name_list => (waiting_room));
```

Application:

```
wait (clerk,waiting_room);
hold(uniform(30.0,50.0));
release (clerk);

if status (clerk) = busy and then
  average_residence_time(waiting_room) > 50.0
                                          then
      outlet (exit_door);
else
    wait (clerk,waiting_room);
end if;
```

Usage of the SNA's:

```
if status (clerk(3)) then ....;
put (preemption_count(clerk));
```

(2) Resource of 100 freight_cars; two places where entities can wait.

Declaration:

```
freight_car : resource
  ( capacity => 20,
    queue_name_list => (station_1, station_2),
    select_criterion => cyclic);
```

Application:

```
wait (freight_car, at_queue => station_1);
wait (freight_car, at_queue => station_2,
          units_requested => 5);
release (freight_car,5);
```

(3) This example demonstrates the use of the fact, that an entity has not to wait for the service on seizing the server: A car is picked up from a big service station. This requires the following actions: the invoice has to be issued (three cashiers) and the car has to be brought from the parking lot (two man). Both actions can be done independently.

```
cashier: service
   ( capacity => 3,
     queue_name_list => (waiting_cashier));

valet_driver: service
   ( capacity => 2,
     queue_name_list => (waiting_valet));
```

Application: The relevant part of the network:

```
request (cashier,waiting_cashier,
         uniform(5.0,15.0));
request (valet_driver,waiting_valet,
         uniform(10.0,15.0));
wait_end_service ((cashier,valet_driver));
```

5. MODEL DEVELOPMENT

The network or transaction flow part of the Ada Simulation Support Environment differs from the rest of the low level support packages. The degree of abstraction reached by a construct like "server" is so high, that rather complicated Ada constructs are used to implement this. Very little about Ada syntax and semantic has to be known to write rather big discrete event oriented simulation models. In principle the knowledge about how to define variables, how to construct expressions, how to write statements and how to compile Ada programs is sufficient. Generic packages and procedures and instances of those are used intensively in the ASSE network part. This would require quite sophisticated knowledge about these constructs in cases, where the user addresses rather simple actions like to define a service process or to wait for a server.

In this case a specific simulation language has advantages over even the best designed package, because such a language allows to formulate a model (if it fits in the world view of the language) in an absolute straightforward form. A natural way to combine the advantages of Ada and the advantages of a specific, suitable notation should be a language derived from Ada. This means to define a superset of (possibly a subset of) Ada. But both, supersetting or subsetting of Ada, is decidedly against the intentions of Ada. My personal view is, that an extension of Ada in form of a preprocessor together with a standard package for simulation (in analogy to the package STANDARD for Ada) is very desirable, legal (because it would define a new language, not Ada) and even if it would not be legal, it could not be prevented. We call this the ASSE Network Language.

A similar, but slightly different approach would be an interactive system which allows a user to specify his model in form of a dialogue. Then an Ada program is build out of these specifications by that interactive system.

A negative effect for both versions is that a user is sometimes confronted with obscure error messages from the (for him invisible) Ada compilation, a well known effect with preprocessors.

Example: Inspection Station on a Production Line

We use this example to demonstrate some aspects of the possibilities of the network part of the Ada Simulation Support Environment.

The original example is from Schriber (1974). A Q-GERT version of the example can be found in Pritsker (1977), a SLAM version in Pritsker (1979).

"Assembled television sets move through a series of testing stations in the final stage of their production. At the last of these stations, the vertical control setting on the sets is tested. If the setting is found to be functioning improperly, the offending set is routed to an adjustment station, where the setting is modified. After adjustment, the television set is sent back to the last inspection station, where the setting is again inspected. Television sets passing the final inspection phase, whether the first time or after one or more routings through the adjustment station, pass on to a packing area. ... Two inspectors work side_by_side at the final inspection station. ..."

The implementation of the model in ASSE network form, using the ASSE Network Language.

```
model body TV_inspection_and_adjustment is

   entity_name is (tv_set);

   last_station_exit : termination;

   model last_station is
      enter entrance;
      outlet last_station_exit;
   end last_station;

   arrival : creation ( entity_kind => tv_set,
     initialization => tv_set_initialization,
     enter_name => last_station.entrance,
     time_between_creations => uniform(3.5,7.5));

   model body last_station is

      waiting_sets : queue;
      waiting_rejects: queue;

      inspector : server
         ( capacity => 2,
           queue_name_list => (waiting_sets) );

      adjustor : server
         ( capacity => 1,
           queue_name_list => (waiting_rejects) );

   network

      accept entrance;
      loop
         service (inspector, waiting_sets,
                  uniform(6.0, 12.0) );
         if uniform < 0.15 then
            service (adjuster, waiting_rejects,
                     uniform (20.0, 40.0) );
         else
            exit;
         end if;
      end loop;

      outlet (last_station_exit);

   end last_station;

simulate

   initialize (0.0, 480.0);
   trace ( first_execution => 0.0,
           last_execution  => 60.0,
           trace_list => (node_arrivals,
                          entity_sna));
   start_simulation;

end TV_inspection_and_adjustment;
```

Discussion:

We have renounced intentionally all comments in our program. We believe, that the code presented above is selfdocumenting and more easily readable than the corresponding GPSS or SLAM code. We concede that our code contains more verbiage. But this is in accordance with the fact that program code is read a hundred times more often than it is written. The additional verbiage helps the reader essentially; it costs the writer only minimal time.

6. MODEL DEVELOPMENT USING THE INTERACTIVE SYSTEM

It is not easy to reproduce an interactive dialogue in printed form. We show here only two of the screen forms. Places, where user input is possible are indicated by underlined fields; an actual input is surrounded by a box.

6.1 Declaration of a Queue

```
        Ada Simulation Support Environment
              *** Model Design ***
                * Network Part *

                Queue  Declaration

Queue name    | waiting_rejects |

Capacity          32767      (1..integer'last)

Initial occupancy    0       (0..integer'last)

Ranking           FIFO       (FIFO,LIFO,HVFO,LVFO)

Block servers when full _____________________
                        _____________________
                        _____________________
                        _____________________
                        _____________________
```

The input of this form is done in three steps:

1) The name of the queue is entered.

2) In the second part one has to enter:
   - The ranking of the entities in the queue. The default value is FIFO.
   - The maximum number of entities which can be held in the queue. The default value is integer'last.
   - The initial occupancy: how many entities shall be in the queue when the simulation starts. The default value is 0.

3) In the third part of the form one enters the names of the servers who shall be blocked when the queue is full. This list is open ended.

6.2 Declaration of a Server

The input of this form (see next page) is done in three steps:

1) The name of the server is entered.

2) The preempt flag can be set. The default value is NO. In our case the server is not preemptable.
   The default number of parallel servers is 1.

3) If entities can wait in different queues for a server, one has to indicate the select criterion; then the list of queue names is entered. This list is open ended.

```
        Ada Simulation Support Environment
             *** Model Design ***
               * Network Part *

            Server/Resource Declaration

Server name  adjustor

Preemptable                    NO  (YES,NO)
Number of parallel servers      1  (1..integer'last)

Related queues:

   Select criterion:

   _ given order     _ cyclic,
   _ random          _ assembly mode option
   _ largest         _ smallest average number
   _ longest         _ shortest waiting time
   _ largest         _ smallest queue
   _ largest remaining capacity
   _ smallest remaining capacity

   List:
          waiting_rejects
```

7. INTERACTIVE TESTING OF MODELS

The model can be executed in an interactive mode. In this case the model is under control of the model verification package. The trace feature can be switch on and off at any time. In that case all variables can be displayed and changed; all processes (queues, servers etc.) can be watched. The trace can be controlled via display conditions. The output is organized like in a Smalltalk environment. A user can form the screen layout according to his intentions, composing it from different watch-forms, provided by the interactive model verification package. We are showing here the follow up of the server and the queue, defined above.

7.1 Watching a Server

Definition of the display condition:

```
        Ada Simulation Support Environment
             *** Model Verification ***
               * Control  Section *

                  Server/Resource

Server name  adjustor

Display condition:

_ after  1  steps
_ after executing action _____
_ after arrival of entity _____ id _____
_ after arrival of _____ entities _____ id ____
_ when simulation time is ___ in _____ .. _____
_ break on keyboard-interrupt

Please fill out!
```

The name of the server and the display conditions are entered. If one of the conditions is true, the simulation run is interrupted and the watch form for the server is updated.

The display conditions:

1) The system halts after a number of steps (any action related to the indicated server).
2) The system halts after executing a specific action (SERVICE, WAIT, ALTER etc.).
3) The system halts after arrival of an entity with a specific name and a specific id. If no id is given, the system halts after the arrival of any entity with the specific name.
4) The system halts after the arrival of a specified number of entities. The rules are the same as in 2).
5) The system halts when the simulation time is in the given range. If a NOT is typed in the first field, the system halts, when the simulation time is NOT in the given range.
6) The simulation is interrupted, when the break key is pressed on the keybord.

The Watch Form for a Server

```
        Ada Simulation Support Environment
             *** Model Verification ***
                   * Report *

                  Server/Resource

Server name:        ADJUSTOR____

Preemptable:        FALSE
Number of related queues _____1
Select criterion    CYCLIC_____

Action requested:   SERVICE_____
Involved entity
Name: TV_SET_____ Id: _____73

The standard numerical attributes:

Status: BUSY_____      Preempted: FALSE_____
Count: _____72         Preemption count:
Capacity                          _____0
Current: _____1        Remaining: _____0
Processing time
Consumed: _____21.78   Remaining: _____5.11
Utilization:           Average holding time
_____0.93044           _____29.40

Please acknowledge _
```

The watch form for the entity can be displayed together with the watch form for the server, if detailed information about the entity is desired.

7.2 Watching a Queue

The follow up of a queue is defined like for a server/resource process. A similar form is used for the the display conditions. A range for the queue size can be given additionally:

_ when queue size is ___ in _____ .. _____

An optional NOT can be typed in the first field.

The Watch Form for a Queue

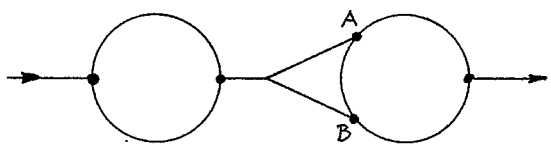
```
      Ada Simulation Support Environment
          *** Model Verification ***
                  * Report *

                     Queue

Queue name:  WAITING SETS_____________

Capacity:                      _____32767
Number of blockable servers:   _________0

Action requested:  INSERT_____
Involved entity
Name: TV_SET___________ Id: _________73

The standard numerical attributes:

Size: _____________3     Full:  FALSE_______

Content
Minimum: _________0      Maximum:  _________3

Count:   ________57      Average residence time
                          _______________36.22
```

8. EXTENSION OF THE MODEL

The example deals obviously only with a part of a greater model. This meets our intention to demonstrate the modular structure of the ASSE network part.

We introduce the attribute 'model_type' for the entity 'tv_set' and we include in the model a packing area. This area can be worked out later in detail following the top-down design principle. For the global model it is important that the packing area disposes of two entrances and one exit. Entrance 'A' is for tv_set of type 'normal', entrance 'B' for type 'de_luxe'. The choice of the right entrance has to be performed in the global model.

The basic structure is:



Arrivals from production — Last station — Packing area — Departure

8.1 Modification of the Model Last Station

It is necessary to alter slightly the original model, since it was not designed to be used in form of a submodel. First we separate and extend the definition of the entity.

```
model entity_description is
   type category is (normal, de_luxe);
   type entity_name is (tv_set);
   type attributes (kind : entity_name ) is
      record
         info: hidden_info;
         model_type: category;
      end record;
      procedure tv_set_initialization;
end entity_description;

model body entity_description is
   procedure tv_set_initialization is
   begin
      if uniform < 0.25 then
         current_entity.model_type:=de_luxe;
      else
         current_entity.model_type:=normal;
      end if;
   end tv_set_initialization;
end entity_description;
```

This submodel is obviousely degenerated; it contains only the definition of the entity. But it can be compiled separately (after being processed by the ASSE-network-preprocessor), and it can be used in other models via a context clause ('with entity_description').

We specify the submodel 'last_station', using 'entity_description'.

```
with entity_description; use entity_description;
model last_station is
   enter entrance;
   outlet last_station_exit;
end last_station;
```

This can be compiled separately.

We compile the model body 'last_station', taking the unchanged code from the original example.

We have to augment the model 'TV_inspection_and_-adjustment' with the context clause and liberate it from the code already present in 'entity_description' and 'last_station'. Then we can simulate the model 'last_station' in this modified form:

```
with entity_declaration, last_station;
use  entity_declaration, last_station;
model body TV_inspection_and_adjustment is

   last_station_exit : termination;

   arrival : creation ( entity_kind => tv_set,
     initialization => tv_set_initialization,
     enter_name => last_station.entrance,
     time_between_creations => uniform(3.5,7.5));

simulate

   initialize (0.0, 480.0);
   trace ( first_execution => 0.0,
           last_execution  => 60.0,
           trace_list => (node_arrivals,
                          entity_sna) );
   start_simulation;

end TV_inspection_and_adjustment;
```

This decomposition has two major advantages: the code is easy to catch and changes of the model body require only a recompilation of that compilation unit.

8.2 Packing Area

Having performed these changes we can come back to

the global model. We have to write first the model specification and the model body for the packing area. We give here only the model specification:

```
with entity_description; use entity_description;
model packing is
   enter entrance_A;
   enter entrance_B;
   outlet packing_exit;
end packing;
```

After sucsessful compilation of both units we can write the global model.

8.3 Global Model

```
with entity_description, last_station, packing;
use  entity_description, last_station, packing;
model body global is

   model TV_inspection_and_packing is

      enter arrivals_from_production;
      enter back_from_last_station;
      enter back_from_packing;
      outlet shipping;

   end TV_inspection_and_packing;

   arrival : creation  ( entity_kind => tv_set,
    enter_name => TV_inspection_and_packing.
                  arrivals_from_production);
    time_between_creations => uniform(3.5, 7.5));

   shipping : termination;

   last_station_exit:
      link (back_from_last_station);
    packing_exit: link (back_from_last_station);

   model body TV_inspection_and_packing is

   network

      accept arrivals_from_production;

      outlet (last_station.entrance);
      accept back_from_last_station;

      if current_entity.model_type=normal then
         outlet (packing.entrance_A);
      else
         outlet (packing.entrance_B);
      end if;
      accept back_from_packing;

      outlet(shipping);

   end TV_inspection_and_packing;

simulate

   initialize (0.0, 480.0);
   trace (  first_execution => 0.0,
            last_execution  => 60.0,
            trace_list => (node_arrivals,
                           entity_sna) );
   start_simulation;

end global;
```

REFERENCES

Adelsberger, H.H., "ASSE - Ada Simulation Support Environment", Proceedings of the 1982 Winter Simulation Conference, San Diego 1982.

Adelsberger, H.H., "A Structured and Modular Approach to Transaction Flow Models", Proceedings of the 1983 Summer Computer Simulation Conference, Vancouver 1983.

Adelsberger, H.H., "Modeling and Simulation in Ada", Proceedings of the 1st. European Simulation Congress, Aachen 1983.

Adelsberger, H.H., "Transaction Flow Models in Ada: Technical Background", Proceedings of the Conference 'Simulation in Strongly Typed Languages: Ada, Pascal, Simula', San Diego 1984.

Brayant, R.M., "Discrete System Simulation in Ada", Simulation Vol. 39, No. 4, Oct. 1982.

Dahl, O.J., Myhrhaug, B. and Nygaard, K., Simula 67 Common Base Language, Norwegian Computing Center, Oslo, 1970.

Franta, W.R., A Process View of Simulation, Elsevir North-Holland, 1977.

Henriksen, J. O., "GPSS - Finding the Appropriate World-View", Proceedings of the 1981 Winter Simulation Conference, Atlanta 1981.

Henriksen, J. O., GPSS/H User's Manual, Second Edition, Wolverine Software Corporation, Annandale 1983.

Henriksen, J. O., "State-of-the-Art GPSS", Proceedings of the 1983 Summer Computer Simulation Conference, Vancouver 1983.

Kreutzer W., "'Of Clouds & Clocks' - A Guided Walk through a Simulator's Tollbox", NZOR Volume 11 Number 1, pp. 51-119, January 1983.

Lomow, G. and Unger, B., "The Process View of Simulation in Ada", Proceedings of the 1982 Winter Simulation Conference, San Diego 1982

Pritsker, A.A.B., The GASP IV Simulation Language, John Wiley & Sons, 1974.

Pritsker, A.A.B., Modeling and Analysis Using Q-GERT Networks, John Wiley & Sons, 1977.

Pritsker, A.A.B. and Pegden C.D., Introduction to Simulation and SLAM, John Wiley & Sons, 1979.

Schriber, T.J., Simulation using GPSS, John Wiley & Sons, 1974.

Shub, Ch.M., "Discrete Event Simulation Languages", Simulation with discrete Models: A State-of-the Art View, Winter Simulation Conference 1980, University of Ottawa, 1980.

Unger, B., "Programming Languages for Computer System Simulation", Simulation, Vol. 30, No. 4, Apr. 1978.

United States Department of Defense, Reference Manual for the Ada Programming Language, Washington, 1983.

United States Department of Defense, Requirements for Ada Programming Support Environments - 'Stoneman', Washington, 1980.

Zeigler, B.P., Theory of Modelling and Simulation, John Wiley & Sons, New York 1976.